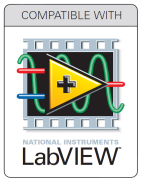
Smartphone & Cross-platform Communication Toolkit

User Manual



Release 3.0

July 2013 Edition

Worldwide technical support and product information:

www.toolsforsmartminds.com

TOOLS for SMART MINDS Corporate headquarter

Via Padania, 16 Castel Mella 25030 Brescia (Italy)

Copyright © 2010 Tools for Smart Minds. All rights reserved.

Contents

Figure index 8

ABOUT THIS MANUAL 11

Conventions 11

INTRODUCTION 13

Overview 13

Top reasons to use Smartphone & cross platform Communication Toolkit (SCCT) 13

SCCT application examples 14

Communicating data across a network, on a complex assembly line 14

Publishing real time data to mobile devices 14

Delivering high quality maintenance service to your customer 15

Supported platforms 15

publisher library 15

Subscriber library 15

Installing the Smartphone & Cross-platform Communication Toolkit 16

GETTING STARTED WITH THE SMARTPHONE & CROSS-PLATFORM COMMUNICATION TOOLKIT 19

Communication concepts 19

Smartphone & Cross-platform Communication Toolkit's Palette 21

Using the Smartphone & Cross-platform Communication Toolkit 22

Publisher Class 23

Creating a Publisher 24

Publishing analog data 25

Defining a system configuration 26

Managing Welcome Kit 26

Publishing digital data 27

Subscriber Class 29

Creating a Subscriber 31

Starting and stopping data transmission 31

Sending Messages 33
Receiving Messages 35
How to check the state of your connection 35
Closing communication 35
CREATING AND MANAGING ALERTS AND USER REQUESTS 36
Overview 36
Reading available messages 36
Sending message to a specific subscriber 36
Managing User Messages 37
Notifying an error 37
SOURCE SELECTION 38
Enabling source filter 39
Selecting Source list 39
Sending a filter command 40
Managing broadcasting messages to by-pass selected source filter 42
PUBLISHING AND RECEIVING IMAGES 43
Publishing an image (example 1) 43
Publishing an image (example 2) 43
Receiving an image (example 1) 44
PUBLISHING AND RECEIVING 2D ARRAYS 45
PUBLISHING AND RECEIVING FILES 45
File exchange example 1 45
File exchange example 2 46
ADVANCED OPTIONS 46
Controlling active connections 46
Reading active connection count 47
Reading active connection addresses 47
Reading active subscribers profiles 47

Managing events 48
Changing API-Key at run-time 50
Adding a new API-Key 50
Removing an existing API-Key 51
Changing max DT 51
writing and reading safety mode 51
Publishing and receiving custom data (supported only by LabVIEW applications) 53
Publishing and receiving custom XML data 54
Publishing and receiving server location 55
BUILD SPECIFICATIONS **57**
APPENDIX A – ERROR TABLE **59**
Publisher error codes 59
Subscriber error codes 60
INDEX **62**

FIGURE INDEX

FIGURE 1–SMARTPHONE & CROSS-PLATFORM COMMUNICATION TOOLKIT'S PACKAGE INFORMATION PAGE. 17

FIGURE 2– PACKAGE INFORMATION PAGE WHEN UPGRADING FROM PREVIOUS VERSION ON SMARTPHONE & CROSS-PLATFORM COMMUNICATION TOOLKIT. 17

FIGURE 3– SMARTPHONE & CROSS-PLATFORM COMMUNICATION TOOLKIT'S LICENSE AGREEMENT PAGE 18

FIGURE 4– AFTER INSTALLATION, A SUMMARY PAGE SHOWS PACKAGE STATUS. 18

FIGURE 5 - COMMUNICATION SCHEME OF SCCT 19

FIGURE 6– HOW DATA PACKAGES ARE ORGANIZED AT SUBSCRIBER SIDE. 20

FIGURE 7– AN APPLICATION CAN PROCESS INCOMING PACKAGES ACCORDING TO THEIR DATA TYPES. 20

FIGURE 8- SMARTPHONE & CROSS-PLATFORM COMMUNICATION TOOLKIT'S PALETTE. 21

FIGURE 9- SIMPLE PUBLISHER EXAMPLE WITH ANALOG DATA. 25

FIGURE 10 - USE OF THE WELCOME KIT WITH A CONFIGURATION CLUSTER. 27

FIGURE 11-SIMPLE PUBLISHER EXAMPLE WITH DIGITAL DATA. 27

FIGURE 12- SIMPLE SUBSCRIBER EXAMPLE THAT READS ANALOG DATA ONLY 31

FIGURE 13 – TRANSFERSTATUS.VI DETERMINES THE FLOW OF PACKETS BETWEEN PUBLISHER AND SUBSCRIBER. 31

FIGURE 14- SIMPLE SUBSCRIBER EXAMPLE THAT READS DIGITAL DATA ONLY. 32

FIGURE 15- SIMPLE SUBSCRIBER THAT READS CONFIGURATION CLUSTER BEFORE READING ANALOG DATA. 32

FIGURE 16–PUBLISHDATA_MESSAGE.VI EXAMPLE. 34

FIGURE 17–MESSAGE SENT BY PUBLISHER CASE IN BROADCAST MODE (LEFT) AND TO A SINGLE CLIENT (RIGHT). 34

FIGURE 18- READING A MESSAGE FROM PUBLISHER. 35

FIGURE 19- USING PROPERTIES NODE AND READCONNEDCTED.VI TO CHECK CONNECTION STATUS. 35

FIGURE 20- USING *AVAILABLE MESSAGE COUNT* PROPERTY NODE TO CHECK IF SUBSCRIBER'S MESSAGES ARE PRESENT. 36

FIGURE 21- SENDING MESSAGE TO A SPECIFIC SUBSCRIBER WITH PUBLISHDATA_MESSAGE.VI. 36

FIGURE 22- MANAGING RECEIVED MESSAGE EXAMPLE 37

FIGURE 23- NOTIFYING LABVIEW ERROR TO ALL SUBSCRIBERS WITH NOTIFYERROR.VI. 37

FIGURE 24 – SCENARIO WITH A MULTIPLE DATA SOURCE SERVER AND CLIENTS THAT NEED A SUBSET OF PUBLISHED DATA. 38

FIGURE 25- SET SOURCEFILTERENABLED=TRUE WHEN YOU PLAN TO TAKE ADVANTAGE OF DATA FILTERING AT SERVER SIDE. 39

FIGURE 26- SOURCE LIST SELECTION 40

FIGURE 27 – SENDING A FILTER REQUEST TO SERVER. 40

FIGURE 28 – SENDING TWO DIFFERENT FILTER REQUESTS TO SERVER. 41

FIGURE 29 - READING FILTERID FOR EVERY RECEIVED DATA PACKAGE 42

FIGURE 30- BROADCAST TRANSMISSION. 42

FIGURE 31– PUBLISHING AN IMAGE WITH PUBLISHDATA_IMAGE1.VI 43

FIGURE 32– PUBLISHING AN IMAGE STORED ON DISK, WITH PUBLISHDATA_IMAGE2.VI. 44

FIGURE 33– RECEIVING AN IMAGE FROM A CLIENT WITH READ IMAGE.VI. 44

FIGURE 34 - PUBLISHING DIFFERENT TYPES OF 2D ARRAYS 45

FIGURE 35 - READING DIFFERENT TYPES OF 2D ARRAYS 45

FIGURE 36– PUBLISHING A FILE WITH PUBLISHDATA_FILE_1.VI 46

FIGURE 37– PUBLISHING A FILE WITH PUBLISHDATA_FILE_2.VI 46

FIGURE 38- USE READ ACTIVESUBSCRIBERSCOUNT.VI TO CHECK THE COUNT OF ACTIVE CONNECTIONS 47

FIGURE 39- USE READ ACTIVESUBSCRIBERSADDRESSES TO GET INFO ABOUT ACTIVE SUBSCRIBERS. 47

FIGURE 40 - USE PROPERTY NODES TO GET INFORMATION ABOUT ACTIVE SUBSCRIBERS. 47

FIGURE 41- USE READ SUBSCRIBERPROFILE.VI TO RETRIEVE INFORMATION ABOUT AN ACTIVE CONNECTION. 48

FIGURE 42 – ENABLING THE REQUEST FIFO WITH ENABLEEVENTLOGGING PROPERTY NODE 49

FIGURE 43 – THIS EXAMPLE SHOWS HOW TO MANAGE "AUTHENTICATION SUCCESSFUL" EVENT. 50

FIGURE 44 – THIS EXAMPLE SHOWS HOW TO MANAGE THE CASE A CONNECTION HAS BEEN CLOSED BY CLIENT. 50

FIGURE 45 – ADDING API-KEY TO ACTIVE API-KEY SET.VI 51

FIGURE 46 – REMOVING AN API-KEY FROM ACTIVE API-KEY SET.VI. 51

FIGURE 47 – CHANGING MAXDT TO CHECK SUBSCRIBER'S SYSTEM TIME. 51

FIGURE 48 – WRITING AND READING SAFETY MODE 52

FIGURE 49–PUBLISHER APPLICATION AND SUBSCRIBER APPLICATION USE CODE FIELD TO IDENTIFY THE PROPER DATA TYPE. BOTH APPS HAVE TO KNOW THE DATA TYPE ASSOCIATED TO THE CODES. 53

FIGURE 50 – WITH PUBLISHER, USE PUBLISHDATA.VI METHOD TO SEND XML DOCUMENTS. 54

FIGURE 51 – WITH SUBSCRIBER, USE SENDCUSTOMXMLDATA.VI TO SEND XML DOCUMENTS TO PUBLISHER. 54

FIGURE 52 – WITH PUBLISHER, USE READ CUSTOMXMLDATA.VI TO PROCESS XML PACKETS. 54

FIGURE 53 – WITH SUBSCRIBER, USE READ CUSTOMXMLDATA.VI TO PROCESS XML PACKETS 55

FIGURE 54 – WITH PUBLISHER AND SUBSCRIBER YOU HAVE DIFFERENT METHODS AND PROPERTY NODES TO GET THE COUNT OF AVAILABLE XML UNPROCESSED DOCUMENTS. 55

FIGURE 55 – USE UPDATELOCATION.VI TO SEND CURRENT PUBLISHER POSITION TO CLIENTS 55

FIGURE 56 – IN SUBSCRIBER, USE SENDCUSTOMXMLDATA.VI TO SEND XML DOCUMENTS TO PUBLISHER. 56

FIGURE 57 – IN SUBSCRIBER, USE READ AVAILABLELOCATIONCOUNT.VI TO VERIFY THE NUMBER OF UNPROCESSED LOCATION PACKAGES. 56

FIGURE 58 – "ADDITIONAL EXCLUSIONS" PAGE MUST BE PROPERLY CONFIGURED. 57

FIGURE 59 – "BUILD ERRORS" WITH ERROR CODE 1562. 58

About this Manual

The Smartphone &Cross-platform Communication Toolkit User Manual describes the virtual instruments (VIs) used to communicate and pass data between LabVIEW and either a local or remote application. You should be familiar with the operation of LabVIEW, your computer and your computer operating system.

Conventions

The following conventions appear in this manual:

▸	The ▸ symbol leads you through nested menu items and dialog box options to a final action. The sequence **Tools** ▸ **Options** directs you to pull down the **Tools** menu, select **Options** item.
Bold	Bold text denotes items that you must select or click on the software, such as menu items and dialog box options. Bold text also denotes parameter names.
italic	Italic text denotes variables, emphasis, a cross reference, or an introduction to a key concept. This font also denotes text that is a placeholder for a word or value that you must supply.
`monospace`	Text in this font denotes text or characters that you should enter from the keyboard, sections of code, programming examples, and syntax examples. This font is also used for the proper names of disk drives, paths, directories, programs, subprograms, subroutines, device names, functions, operations, variables, filenames and extensions, and code excerpts.
`monospace italic`	Italic text in this font denotes text that is a placeholder for a word or value that you must supply.

INTRODUCTION

This chapter describes the installation procedure, installed components, and the main features of the Smartphone & Cross-platform Communication Toolkit.

OVERVIEW

The Smartphone & Cross-platform Communication Toolkit is an add-on package for communicating data trough applications. The toolkit contains a set of high level functions for sending your application data and advanced functions for customized tasks.

The following list describes the main features of the Smartphone & Cross-platform Communication Toolkit.

- Works over any TCP/IP connection
- Works over Local Area Networks as well as Internet connections.
- Implements the publisher – subscriber pattern (also known as Observer pattern)
- Authenticates subscribers through an API-KEY.
- Controls in background the state of every connection to identify loss of communication.
- Publishes GPS coordinates to manage mobile systems.
- Works with platform independent Data format and communicate with multiple platforms at the same time: third party vendors have implemented toolkit to develop on Android platform, Java, .NET and VB, Unix and iOs.

TOP REASONS TO USE SMARTPHONE & CROSS PLATFORM COMMUNICATION TOOLKIT (SCCT)

Adopting this toolkit you have the following advantages:

Simplify communication: don't care about communication details over a TCP communication channel, SCCT does it for you.

Multiple platforms are supported: exchange your data with a protocol supported on a wide range of platforms and programming languages.

It's reliable: many applications have been created with this toolkit around the world.

Speed up your development activity: this toolkit allows the creation of distributed application and let you save a lot of your time.

SCCT has been created by LabVIEW developers for LabVIEW developers; it includes some great features supported on LabVIEW platform only (see Publishing CustomData in this manual) so if you need to exchange data with other LabVIEW applications, take advantage of the power of SCCT to deliver high quality code and reduce developing time.

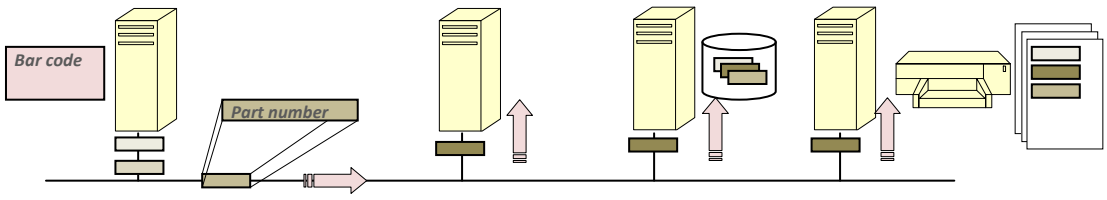
Today applications need to retrieve information from database, bar code readers, OCR systems, remote data acquisition or technical operators with mobile devices. Often you have to create systems capable to exchange data with legacy applications created on different platforms. Every time you have to design an ad hoc communication protocol, code both server and client side routines, debug them. Every time! With SCCT you have a new tool that let you save a lot of time, lets you focus on your project's core and manage multiple communications at the time. In the following examples you are going to see how SCCT can dramatically improve the quality of your software solutions.

SCCT APPLICATION EXAMPLES

SCCT can be successfully applied in many real world situations. In this chapter we discover where SCCT helps developers to succeed to deliver high valued solutions.

COMMUNICATING DATA ACROSS A NETWORK, ON A COMPLEX ASSEMBLY LINE

Creating an application that broadcasts part numbers and barcodes coming from databases or bar code readers, to all computers on an assembly line, as schematized below. On modern assembly lines, many computers control single processing stations and they need to exchange data among them to know, part numbers, print codes and certificates, store test results on different databases. SCCT provides many functionality to exchange data with applications developed with heterogeneous programming languages. Besides, when your customer asks you to show some data on a mobile device, SCCT is capable to communicate with a large variety of smart phones and tables and you do not have to change a single line of code of your software. SCCT cares of all connected devices and communicate your data to all of them at the same time, receives user's requests and organize them in a time-ordered FIFO so that you can process them easily.



PUBLISHING REAL TIME DATA TO MOBILE DEVICES

When you have to publish acquired data in real time, SCCT is the best choice because with few SCCT Vis your applications are supported on a wide range of platforms: Android, iPhone, etc. SCCT is safe because every device must authenticate to your application and you can control who is connected in any time. Using SCCT you deliver open solution to your customers because they can use their favorite mobile platform to connect to your data. And if customers change mobile device family, you don't need to modify your code!

DELIVERING HIGH QUALITY MAINTENANCE SERVICE TO YOUR CUSTOMER

Including SCCT into your existing applications, let you offer fast support to your customers. You and your customers can monitor deployed applications everywhere, with a tablet, phone, and desktop. Integrating SCCT capabilities into existing applications makes easy debug and signal analysis of deployed systems around the world.

SUPPORTED PLATFORMS

SCCT is composed of two main components:

- publisher library
- subscriber library

PUBLISHER LIBRARY

This library let you create a full-featured publisher, which authenticates incoming subscribers, check connection status, sends data to all active publishers and passes their request to your application.

This library is available as a set of Vis for LabVIEW 2010 or later.

To get more details or download an evaluation copy of this library please visit:

http://www.toolsforsmartminds.com/products/SCCT.php

SUBSCRIBER LIBRARY

This library let you create a subscriber which handles all communication details with a publisher so you don't have to. It receives data packages and present them to your application according to their data types.

This library is available for the following platforms and languages:

Name	***Operating System***	***Development Language***
SCCT Subcriber for LabVIEW	Windows	LabVIEW 2010
SCCT Subscriber for Java	Java VM 5.0 or later	Java
SCCT Subscriber for Android	Android 2.1 or later	Java
SCCT Subscriber for Linux	Linux Kernel 2.6.30	ANSI - C
SCCT Subscriber for iPhone/iPad	iOs	Objective C
SCCT Subscriber for Javascript	HTML5 browser	Javascript

To get more details or download your **free** copy of SCCT subscriber library, please visit:

http://www.toolsforsmartminds.com/products/SCCT.php

INSTALLING THE SMARTPHONE & CROSS-PLATFORM COMMUNICATION TOOLKIT

Smartphone & Cross-platform Communication Toolkit is shipped as a VI Package Manager. You can download it from

www.toolsforsmartminds.com/products/SCCT.php

Before installing Smartphone & Cross-platform Communication Toolkit you must install a copy of VI Package Manager on your machine. Youcan get a free copy of VIPM at this address:

http://www.jki.net/vipm/download

To install Smartphone & Cross-platform Communication Toolkit double click open the vip file:

smartphone_&_crossplatform_communication_toolkit-x.x.x.xx.vip

and follow the installation wizard. Package contains LabVIEW Vis as well as documents in PDF format accessible from LabVIEW (**Help ▸ TOOLS for SMART MINDS ▸ SCCT User Guide)**and libraries to create applications with Java, Android and .Net (in *c:\SCCT\cross-platform libs*). Visit http://www.toolsforsmartminds.com to get more details.

Double click on *smartphone_&_crossplatform_communication_toolkit-x.x.x.xx.vip*

or open it from VIPM and the package information page appears, as shown in the next figure.

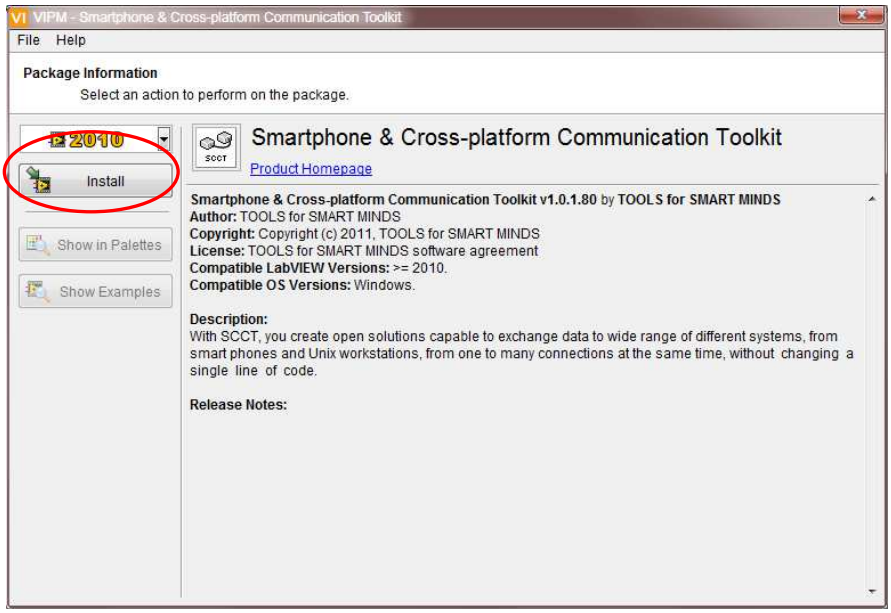

VIPM - Smartphone & Cross-platform Communication Toolkit
File Help
Package Information
Select an action to perform on the package.
2010
Install
Show in Palettes
Show Examples
Smartphone & Cross-platform Communication Toolkit
Product Homepage
Smartphone & Cross-platform Communication Toolkit v1.0.1.80 by TOOLS for SMART MINDS
Author: TOOLS for SMART MINDS
Copyright: Copyright (c) 2011, TOOLS for SMART MINDS
License: TOOLS for SMART MINDS software agreement
Compatible LabVIEW Versions: >= 2010.
Compatible OS Versions: Windows.
Description:
With SCCT, you create open solutions capable to exchange data to wide range of different systems, from smart phones and Unix workstations, from one to many connections at the same time, without changing a single line of code.
Release Notes:

FIGURE 1–SMARTPHONE & CROSS-PLATFORM COMMUNICATION TOOLKIT'S PACKAGE INFORMATION PAGE.

Press "Install" button in the upper left corner, VIPM shows available actions. When upgrading Smartphone & Cross-platform Communication Toolkit to a new version, VIPM displays"upgrade" in Action column.

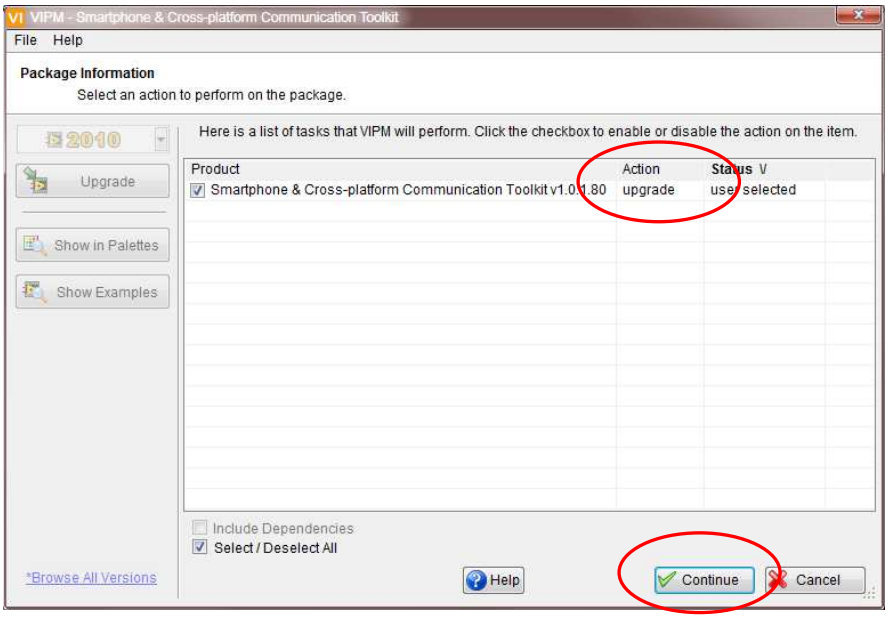


FIGURE 2– PACKAGE INFORMATION PAGE WHEN UPGRADING FROM PREVIOUS VERSION ON SMARTPHONE & CROSS-PLATFORM COMMUNICATION TOOLKIT.

Press "Continue" in the lower right corner to install/upgrade the selected package.

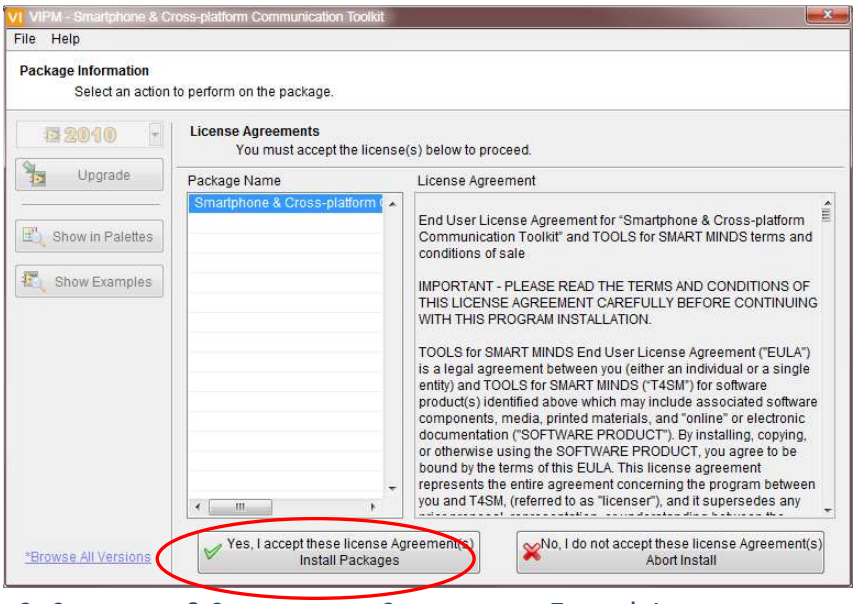

VIPM - Smartphone & Cross-platform Communication Toolkit
File Help
Package Information
Select an action to perform on the package.
2010
Upgrade
Show in Palettes
Show Examples
License Agreements
You must accept the license(s) below to proceed.
Package Name
License Agreement
Smartphone & Cross-platform
End User License Agreement for "Smartphone & Cross-platform Communication Toolkit" and TOOLS for SMART MINDS terms and conditions of sale
IMPORTANT - PLEASE READ THE TERMS AND CONDITIONS OF THIS LICENSE AGREEMENT CAREFULLY BEFORE CONTINUING WITH THIS PROGRAM INSTALLATION.
TOOLS for SMART MINDS End User License Agreement ("EULA") is a legal agreement between you (either an individual or a single entity) and TOOLS for SMART MINDS ("T4SM") for software product(s) identified above which may include associated software components, media, printed materials, and "online" or electronic documentation ("SOFTWARE PRODUCT"). By installing, copying, or otherwise using the SOFTWARE PRODUCT, you agree to be bound by the terms of this EULA. This license agreement represents the entire agreement concerning the program between you and T4SM, (referred to as "licenser"), and it supersedes any
*Browse All Versions
Yes, I accept these license Agreement(s)
Install Packages
No, I do not accept these license Agreement(s)
Abort Install

FIGURE 3– SMARTPHONE & CROSS-PLATFORM COMMUNICATION TOOLKIT'S LICENSE AGREEMENT PAGE.

Before stating installation procedure, you have to accept Smartphone & Cross-platform Communication Toolkit license agreement, as shown above.

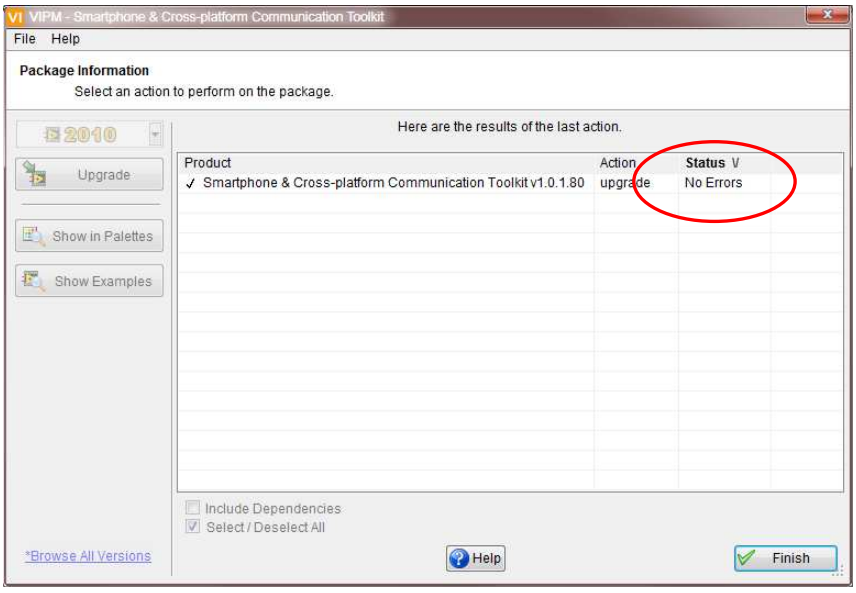


FIGURE 4– AFTER INSTALLATION, A SUMMARY PAGE SHOWS PACKAGE STATUS.

When installation completes, a summary page is displayed. After installation is completed, LabVIEW must be closed and restarted.

GETTING STARTED WITH THE SMARTPHONE & CROSS-PLATFORM COMMUNICATION TOOLKIT

COMMUNICATION CONCEPTS

Smartphone & Cross-platform Communication Toolkit implements a fully bidirectional communication channel between an application identified as server and a pool of applications called clients. Your application manages communication through SCCT's APIs and publish/receive data with high-level functions. SCCT cares of all connected systems and handles data transmission so you don't have to. The clients have to authenticate themselves with a password (in SCCT terminology, it is called API-Key). The following figure represents the communication scheme:

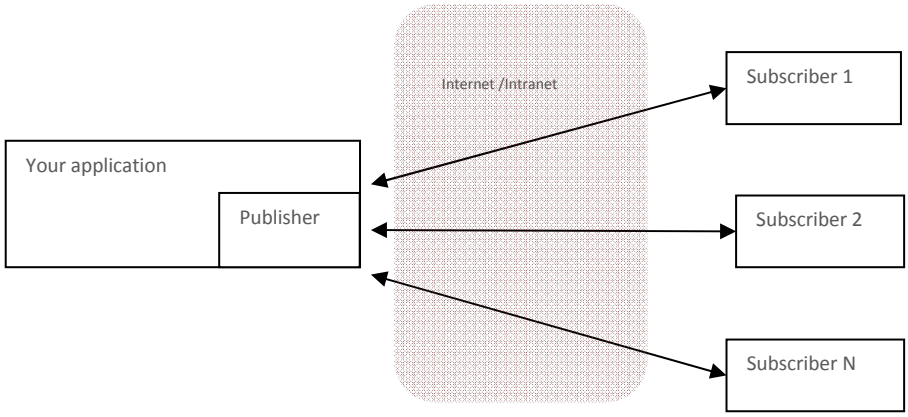


FIGURE 5 - COMMUNICATION SCHEME OF SCCT

When an application wants to receive data, first asks the publisher to be inscribed among the active subscribers. Publisher will accept all incoming requests with the valid API-KEY.

Received data are organised into separated FIFOs so that subscriber application can process data packages according to their types. The following figure shows how data packages are sent and organized on the publisher and subscriber side, respectively. Starting from a unique FIFO queue of different package types, the subscriber side task arranges the received packages in different FIFO structures.

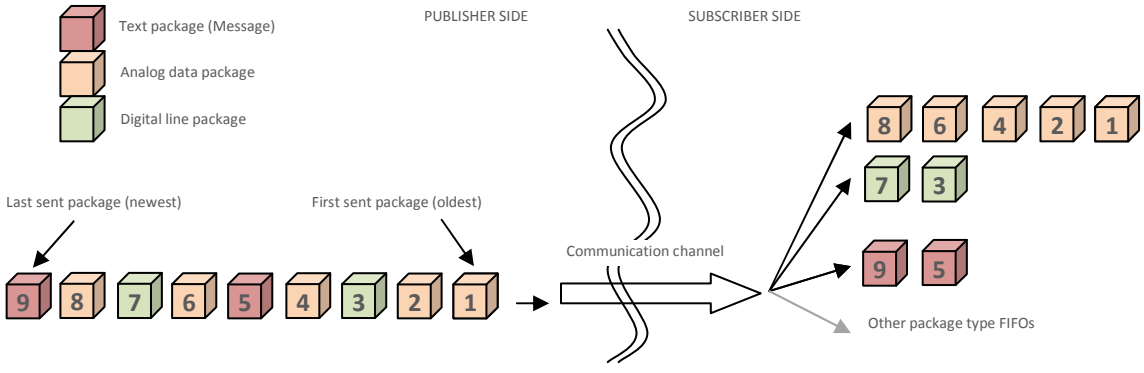


FIGURE 6– HOW DATA PACKAGES ARE ORGANIZED AT SUBSCRIBER SIDE.

In this architecture, packages can be processed very easily with dedicated Vis. In many real life applications the exact order in which data packages are generated is not very important, and with LabVIEW multithread capability you can dedicate specific task to every package type and deliver robust software solution as shown below.

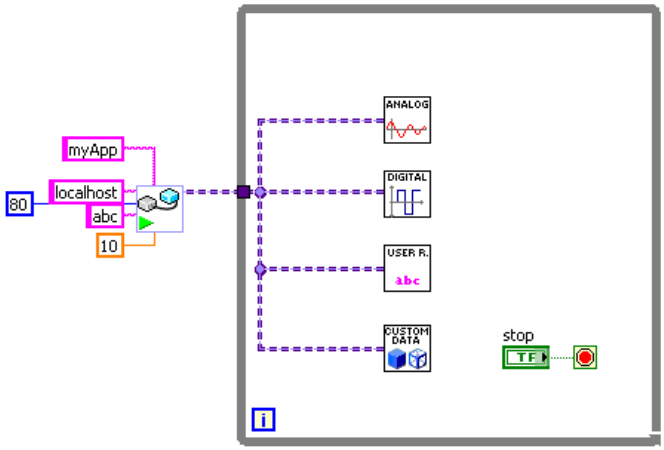


FIGURE 7– AN APPLICATION CAN PROCESS INCOMING PACKAGES ACCORDING TO THEIR DATA TYPES.

Because of the wide range of devices the Smartphone & Cross-platform Communication Toolkit works with, some portability issues remain. Consider the following issues when choosing your way to publish data.

- Some smartphones and tablets use CPU with low computing power so are not able to receive and process large streams of data.
- Smartphone & Cross-platform Communication Toolkit uses a platform independent data format and subscribers require some computing power to decode data streams into their specific binary format.
- Smartphone & Cross-platform Communication Toolkit handles communication with subscribers as a set of peer to peer connections and every data you publish is transmitted individually to each subscriber. So you have to identify the right size of your data streams to avoid band saturation over your communication channel.
- Some data types are not supported on all platforms.

SMARTPHONE & CROSS-PLATFORM COMMUNICATION TOOLKIT'S PALETTE

After you installed the toolkit, you can start using Smartphone & Cross-platform Communication Toolkit from a blank VI. To get access to library's palette, browse through **Functions ▸ Data communication ▸ TOOLS for SMART MINDS**, as shown below.

Publisher VIs have light red background icons, Subscriber VIs have white background icons.

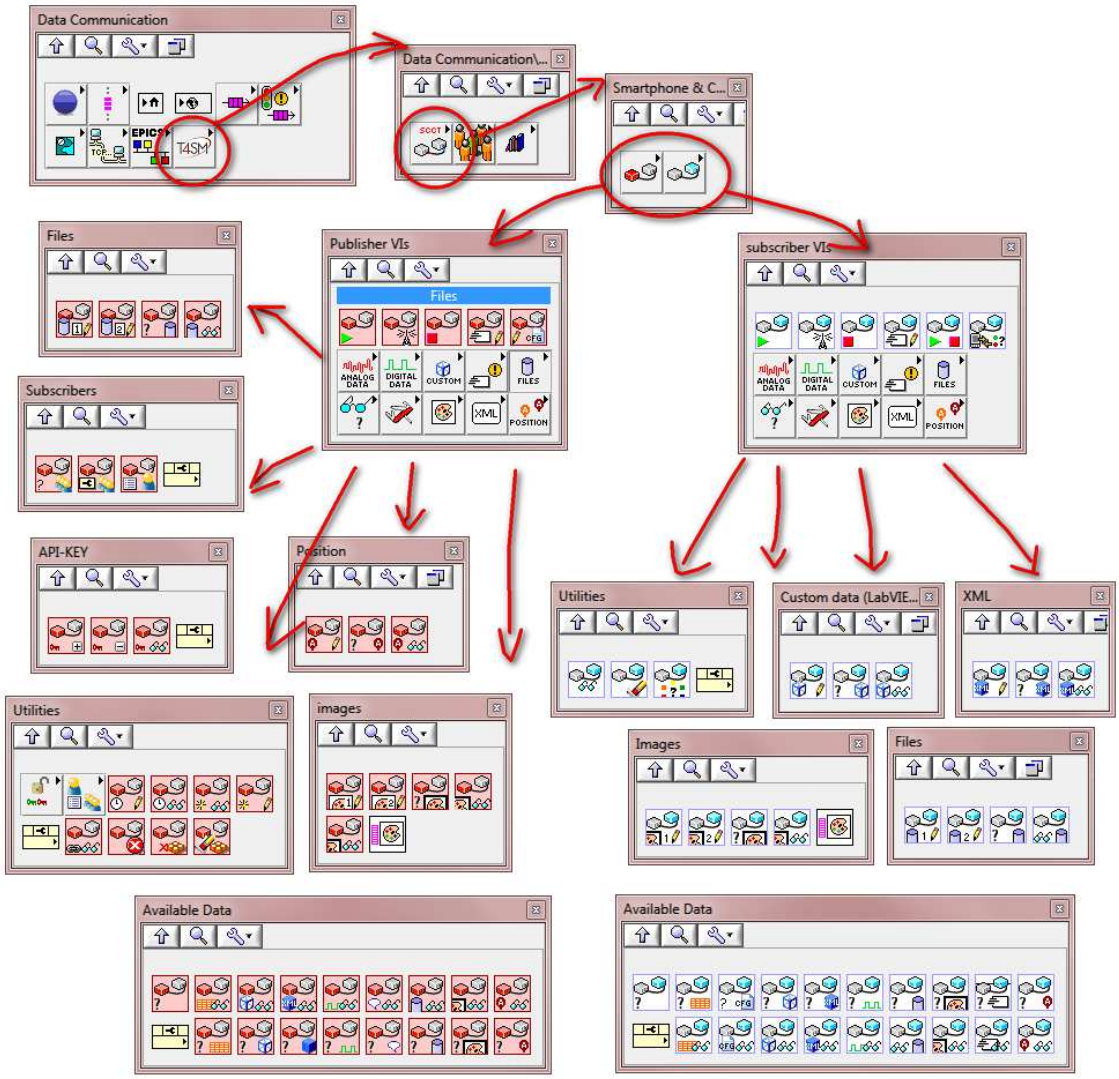


FIGURE 8- SMARTPHONE & CROSS-PLATFORM COMMUNICATION TOOLKIT'S PALETTE.

USING THE SMARTPHONE & CROSS-PLATFORM COMMUNICATION TOOLKIT

This Toolkit is composed of two main components: Publisher that creates the server side of your communication system and the subscriber that implements the client side. Publisher and Subscriber work together to pass data from one application, which holds data to many applications on different systems (MS-Windows OS family, Linux, Apple systems, mobile devices, etc.). Publisher uses a platform independent data format to transmit your data so that all subscribers can read them. In this way, you add a little overhead to a simple transmission that uses binary data format, but gain a great portability and opportunity to communicate with heterogeneous systems. To better understand the way this communication works, consider the following example. A publishing company receives requests from different subscribers that want to receive a magazine. As long as they are subscribed, they receive the magazine. When they don't want to receive it anymore, simply cancel their subscription. Your application can implement more than one Publisher each of them works on different port of your machine. An application can contain publishers and subscribers together, working with different remote machines at the same time.

Either objects work in background of your application with specific tasks that are created and destroyed automatically. This library has been created by LabVIEW developers and includes some advanced features supported only by LabVIEW apps. Please be careful when design your communication solutions so that your data can be properly treated by all subscriber. So when your system is designed to work with LabVIEW programs only, CustomData packages greatly simply data transfer among processes, but cannot be processed by subscribers created, for example, with SCCT for Java.

PUBLISHER CLASS

Publisher is a Class with methods and properties detailed in the following tables.

Publisher Class properties		
Property name	*description*	*Read/write*
availableMessageCount	gets the count of received messages from subscribers	Read only
enableEventLogging	Sets/gets the management of subscriber requests	Read/Write
Port	Gets the actual port number	Read only
API-Key	Gets the actual API-Key	Read/Write
activeSubscribersCount	Gets the count of active connections	Read only
activeSubscribersAddresses	Gets the IP address and port of each active connection	Read only
maxDT	Sets/gets the maximum allowed time difference between publisher time clock and subscribers time click	Read/Write

Publisher Class methods	
method name	*description*
Add API-Key	Add an API-Key. Active connections are not affected by this change
ClearWelcomeKit	Clears welcome kit package list
enableEventLogging	Enables the management of requests
notifyError	Transmit a message to a specific connection or to all active connections with a LabVIEW error code and source.
publishData	Polymorphic function that adapts to input data and publishes a data package.
PublishData_2D_DBL_ARRAY	Publishes analog data as 2D array
PublishData_1D_BOOL_ARRAY	Publishes a 1D array of Boolean
PublishData_BOOL_ARRAY	Publishes a boolean array
PublishData_I16_ARRAY	Publishes an array of word signed integers (I16)
PublishData_I32_ARRAY	Publishes an array of long signed integers (I32)
PublishData_I64_ARRAY	Publishes an array of quad signed integers (I64)
PublishData_SGL_ARRAY	Publishes an array of single-precision floating-point data (SGL)
PublishData_DBL_ARRAY	Publishes an array of double-precision floating-point data (DBL)
PublishData_STRING_Array	Publishes an array of strings
publishData_CUSTOM_DATA	Publishes a custom data. Custom data can be a LabVIEW cluster/object. Only LabVIEW clients can handle this data correctly
publishData__XML_DATA	Publishes an XML document
publishData_File_1	Publishes a file your application has loaded in memory. This function let you create file on a remote platform without saving it on local disk
publishData_File_2	Publishes an existing file. This function loads selected file, computes MD5 and transfer data
publishData_Image_1	Publishes an image present in memory as binary string.
publishData_Image_2	Publishes an image loading it from disk. This function works only with a specific set of image formats
publishData_Message	Publishes a message, composed by a message code and its description
PublishData_NewLocation	Publishes new server GPS location
Read activeSubscriberAddresses	Returns the IP address and TCP port of active subscribers
Read activeSubscriberCount	Returns the count of active subscribers
Read API-Keys	Returns current API-Key list
Read availableAnalogDataCount	Returns the count of analog data packages (2D array of doubles) to be processed
Read availableConfigurationCount	Returns the count of configuration packages to be processed
Read availableCustomDataCount	Returns the count of custom data packages to be processed
Read availableXMLDataCount	Returns the count of unprocessed Custom XML packages.
Read availableData	Polymorphic function that returns the count of unprocessed packages by

	checking the queue related to a specific data type
Read availableDigitalDataCount	Returns the count of digital data packages to be processed
Read availableEventCount	Returns the count of unprocessed events
Read availableFilesCount	Returns the count of unprocessed file packages.
Read availableImageCount	Returns the count of unprocessed image packages
Read availableMessageCount	Returns the count of unprocessed message packages.
Read availableNewlocationCount	Returns the count of unprocessed location packages
Read configuration	Reads a system's configuration package
Read constrainCheck	Return a system's configuration package. Configuration packages refer to a server. This function is available when multiple servers are connected into a network
Read CustomData	Reads a custom data package
Read XMLData	Reads an XML data package
Read DigitalData	Returns a digital data package
Read AnalogData	Returns an analog data package
Read Array	Polymorphic function that returns an array data package
Read Event	Get the first unprocessed event
Read File	Reads a file package.
Read Image	Reads an image and returns a LabVIEW's image data cluster.
Read Image(planData)	Reads an image in its raw format. Image is returned as a binary string regardless of its native format
Read Message	Returns a message package
Read NewLocation	Returns a location package
Read Port	Returns the current TCP port used by Publisher
Read Request	Read the first unprocessed request from one of the active connections
Read SubscriberProfile	Returns the profile(connection info) of selected subscriber
Remove API-Key	Removes an API-Key from API-KEY list. Active connections are not affected by this change
RemovePackageFromWelcomeKit	Removes package with specific tag from welcome kit.
selectedSourceList	Selects the source list when connecting to a data server. This function is available only when multiple server are connected into a network
sendMessage	Transmits a message to a specific connection or to all active connections
startPublisher	Implements a publisher object
stopPublisher	Destroys a Publisher object
updateConfiguration	Transmits a new system configuration to all active connections
updateLocation	Updates server location to all clients

CREATING A PUBLISHER

To create a publisher in your LabVIEW application uses palette **SCCT ▸ Publisher**. The following examples show how to create a publisher in few click.

To create a publisher in your application use **startPublisher.vi.** To create a new Publisher two parameters are required:

- Publisher port, which is the TCP port that Publisher uses to manage all TCP connections;
- API-Key, which is the connection password that subscribers must communicate to publisher to be authenticated.

Take care to use one of the available port of your machine. Some ports are reserved for other common applications like port 21 to FTP, 80 to HTTP and so on. Moreover, you have to check that the chosen port is open on your company firewall.

SCCT Communication with Internet Explorer 9 requires Flash Plug-in. SCCT permits to communicate with Internet Explorer 9 through the boolean input **supportIE9WithFlashPlugin** of **startPublisher.vi**. By default, **supportIE9WithFlashPlugin** is enabled. If it is disabled, application cannot communicate with IE9. SCCT 3.0 supports Internet Explorer 10 and it does not require Flash Plug-in to communicate with SCCT. For this reason, **supportIE9WithFlashPlugin** can be disabled with Internet Explorer 10.

PUBLISHING ANALOG DATA

With SCCT library, publish analog data requires a single Vi, **publishData.vi** which is a polymorphic Vi that adapts to input and allows to send different types of data. Analog data are published as a 2D array of double. Notice that every row of your 2D array contains samples of a single channel. If you are publishing data to mobile devices, remember that computational power of smartphones is far lower than PC and they cannot process and display huge amount of data per second. The following figure illustrates a basic example where analog data coming from an acquisition device are sent to all clients. Publishing rate is the ruled by the way your application collects data from physical device.

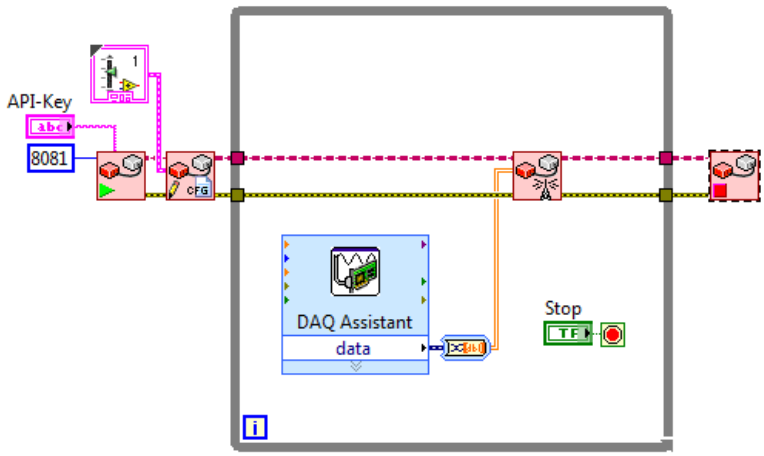


FIGURE 9- SIMPLE PUBLISHER EXAMPLE WITH ANALOG DATA.

In the previous example, a Publisher is created with **startPublisher.vi** that immediately creates all necessary data structures, tasks and takes care of all incoming connections. Before starting to transmit analog data, a system configuration cluster is published through **updateConfiguration.vi**. Please note that configuration cluster must be filled according to the analog signals you want to transmit. Channel configuration must describe each channel of your data acquisition. The system configuration is explained in detail in the following section.

In the while loop your acquired data are published to the active subscribers directly. If no subscribers are connected, data are discarded.

When the loop terminates, **stopPublisher.vi** closes all active tasks and flushes FIFO with user requests.

*Notice: **startPublisher.vi** creates some background tasks which handle the data transfer to and from active subscribers. These tasks are also responsible of checking if connections are lost in case your application is not publishing any information. All these tasks periodically check if your top level VI is running and stop automatically when your top level VI stops.*
***stopPublisher.vi** stops background tasks. When this VI is executed, all active connection are terminated by the publisher and no more connections can be created.*

DEFINING A SYSTEM CONFIGURATION

The system configuration cluster includes several information, such as device name, product type, location, and a detailed description about analog or digital channels. Every analog channel is described by a description, measurement unit, min value, max value, and sampling rate. This information can be used by subscribers to properly adapt graphs and charts. The use of the system configuration package makes sense only when SCCT is used to publish analog and/or digital data. When system configuration changes, updateConfiguration.vi can be used to refresh the configuration to all active subscribers at run time.

Publisher requires a system configuration cluster with at least one analog channel description or one digital line description. If you connect an empty configuration cluster, **startPublisher.vi** rises up an error and doesn't start. The reason of this is behavior because, in past, SCCT was used to publish data from acquisition boards. Now, many new features are available and SCCT can be used to transmit a wide range of different data types, not related to any data acquisition, but configuration constrain remains to keep compatibility with legacy code.

MANAGING WELCOME KIT

With SCCT you can take advantage of Welcome Kit feature. If a new subscriber connects to server after that a data package has already been published, the new subscriber does not receive this data package. To avoid this problem, the Welcome Kit feature can be used. If the Welcome Kit is specified when a data package is published, the publisher sends this data to subscriber when it connects to the server, despite the subscriber was not connected when the data package was sent.

There are many scenarios where it is useful to send some packages to a new subscriber when connection is established. For example, when you publish analog or digital data, a system configuration package should be sent to describe the transmitted channels. If the Welcome Kit is specified in **updateConfiguration.vi,** the publisher sends the system configuration cluster to subscriber when it connects to the server. In this way, the system configuration is the first package that subscriber receives

when it establishes a new connection. The following figure shows how to use the Welcome Kit with the configuration package cluster.

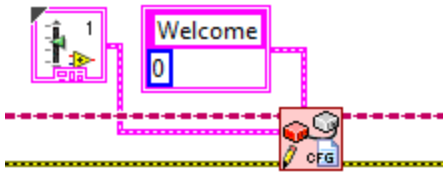


FIGURE 10 - USE OF THE WELCOME KIT WITH A CONFIGURATION CLUSTER.

You can also use the Welcome Kit when you want that new subscribers receive a welcome message when connection is established and after they have to receive server status. SCCT can manage these situations so your application doesn't have to. Whenever you publish some data, you just specify which packages have to be kept in a special set called "Welcome Kit". SCCT identifies packages you wish to add to Welcome Kit by welcomeTag, a string you connect to the proper connector when you send data to clients, as shown below.

PUBLISHING DIGITAL DATA

In this example, a Publisher is created with **startPublisher.vi** that immediately create all necessary data structures and tasks and takes care of all incoming connections. Please note that configuration cluster must be filled according to the digital lines you want to transmit. Configuration must describes each line of your data acquisition. In the while loop your acquired data are published directly to the active subscribers. If no subscribers are connected, data are discarded. When the loop terminates, **stopPublisher.vi** closes all active tasks and flushes FIFO with user requests.

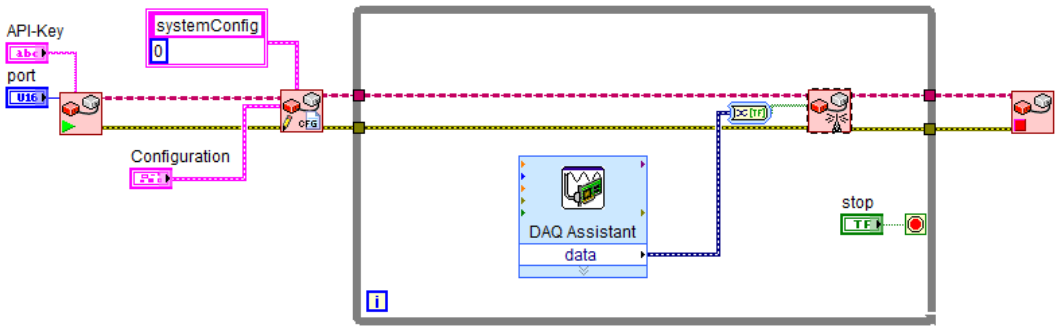


FIGURE 11-SIMPLE PUBLISHER EXAMPLE WITH DIGITAL DATA.

Please note in the above figure that when you start a publisher you create a publisher object with a small set of parameters. The new object will manage by itself most of communication details. If you want to modify the way communications are handled, use LabVIEW property nodes to set desired values to publisher options. The time that occurs between the execution of **publishData.vi** and the effective

transmission of data package to all active subscribers is usually few milliseconds and depends mostly on data size and not on the active connection number.

SUBSCRIBER CLASS

Subscriber is a Class with methods and properties detailed in the following tables:

Subscriber Class properties		
availableMessageCount	Gets the number of received alerts from Publisher that application has to process	Read only
availableAnalogData	Gets the number of analog data packets received from Publisher that application has to process	Read only
availableConfiguration	Gets the number of configuration clusters received from publisher that application has to process. Remember that when you establish a connection, you receive immediately a configuration cluster.	Read only
availableDigitalData	Gets the number of digital data packets received from Publisher that application has to process	Read only
connected	Returns TRUE if connection is active, FALSE if connection is lost	Read only
connectionStatus	Gets a numeric code related to connection status	Read only

Subscriber Class methods	
method name	description
openConnection	Creates a subscriber and open a connection with a running Publisher
closeConnection	Destroys a subscriber and close connection, if active
Read Message	Reads next available alert received from Publisher
Read analogData	Reads next available analog data packet received from Publisher. To activate data transmission, you must use transferStatus.vi
Read digitalData	Reads next available digital data packet received from Publisher. To activate data transmission, you must use transferStatus.vi
Read configuration	Returns the first unprocessed configuration cluster received from Publisher. After a connection is established, Subscriber receives a remote system configuration with a description of analog channels and digital lines, location and other system information.
Read customXMLData	Returns the first unprocessed custom XML data received from Publisher.
Read customData	Returns the first unprocessed custom data cluster received from Publisher. Your application must be able to match the custom data received (a variant value) with the code specified by the application that sends the item.
discardData	Throws all received alerts, analog and digital data and configuration clusters away
transferStatus	Set the status of transmission. If you connect TRUE, publisher start sending analog data and digital lines as soon as they are available on server side. If you connect FALSE, Publisher stop data transmission. Alerts from publisher cannot be stopped.
Send Request	Sends immediately a request to the Publisher.
Send CustomXMLData	Sends immediately a custom XML data to Publisher.
Read availableAnalogDataCount	Returns the count of unprocessed analog data packets.
Read availableDigitalDataCount	Returns the count of unprocessed digital data packets.
Read availableConfigurationCount	Returns the count of unprocessed configuration cluster packets
Read availableMessageCount	Returns the count of unprocessed messages
Read availableCustomDataCount	Returns the count of unprocessed custom data cluster packets
Read availableCustomXMLDataCount	Returns the count of unprocessed custom XML data cluster packets
Read connected	Return a Boolean value with the connection status. If TRUE then connection is active otherwise is FALSE.
Read connectionStatus	Returns a string that indicates the connections status or the failure reason in case openConnection doesn't succeed to establish a valid connection.

Select Source List	Select among tha available a list of sources a sub set of data sources. Server will transmit to client packages that belong to selected sources.

To implement the subscriber in your application and receive data from a source, you must know three parameters:

- Data source address, which is usually the IP address of the machine where Publisher is running on.
- Data source port that is the TCP port of the Publisher.
- API-Key is the key necessary to be authenticated by publisher. If a subscriber uses a wrong API-Key connection is refused by publisher.

When your application succeeds to connect, publisher sends immediately a configuration of remote system i.e. analog channel descriptions, measurement unit, and range of all signals, digital line descriptions and system location GPS coordinates.

CREATING A SUBSCRIBER

In the following example you create a simple subcriber with **openConnection.vi** that needs three parameters: Publisher address (default value is localhost), Publisher port (default value in 8081) and API-Key. Data transmission is started through **transferStatus.vi**.

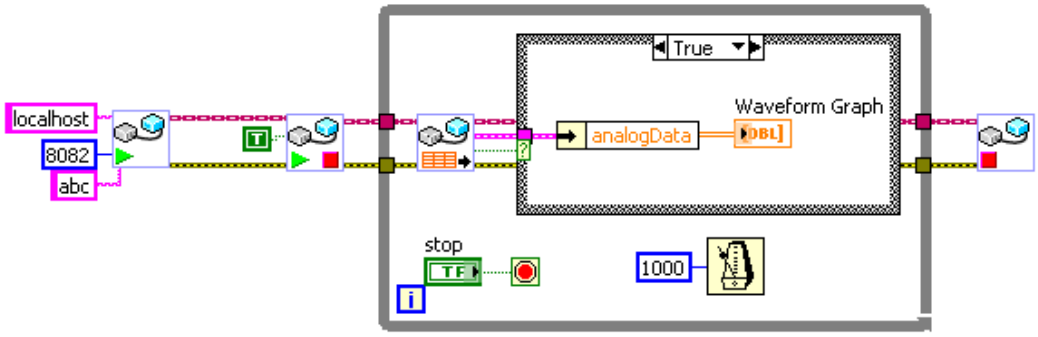


FIGURE 12- SIMPLE SUBSCRIBER EXAMPLE THAT READS ANALOG DATA ONLY.

STARTING AND STOPPING DATA TRANSMISSION

Publisher doesn't start data transmission automatically. After connection, your subscriber must tell to the Publisher to start sending data. If your subscriber doesn't need fresh data, use **transferStatus.vi** with a FALSE constant to tell the publisher to stop sending data. To re-start data transmission use **transferStatus.vi** with TRUE constant.

Publisher doesn't buffer data if transmission is stopped. The following diagram illustrates how SCCT manages data transmission.

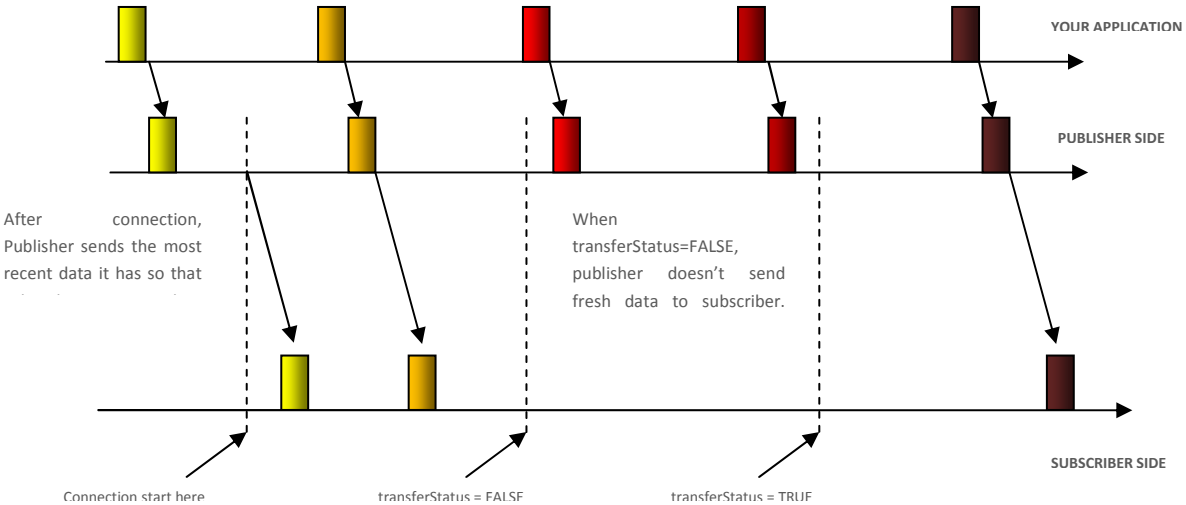

YOUR APPLICATION
PUBLISHER SIDE
After connection, Publisher sends the most recent data it has so that
When transferStatus=FALSE, publisher doesn't send fresh data to subscriber.
SUBSCRIBER SIDE
Connection start here
transferStatus = FALSE
transferStatus = TRUE

FIGURE 13 – TRANSFERSTATUS.VI DETERMINES THE FLOW OF PACKETS BETWEEN PUBLISHER AND SUBSCRIBER.

To get analog data use **Read analogData.vi** that returns a packet of data. Background tasks takes care of all received packets and enqueues them in a FIFO so you can process all packets without data loss. To know the number of available packet use the property node that returns the data packets count. The following figure illustrates the case of reading digital lines only. Notice that at the end of the while loop you always have to close communication with publisher. When communication is closed and you want to open again the communication, use **openCommunication.vi.**

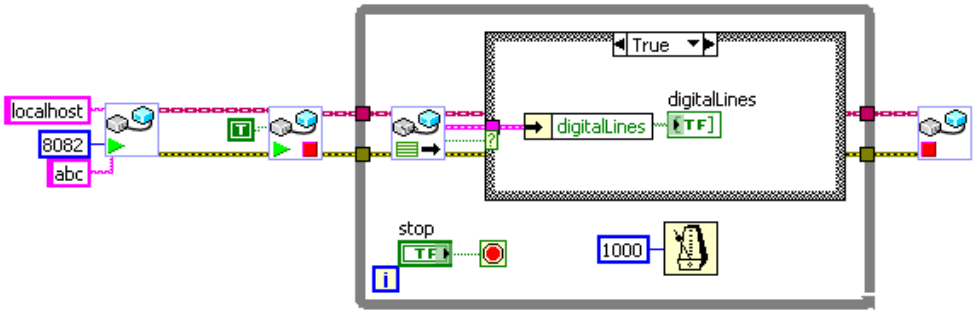


FIGURE 14- SIMPLE SUBSCRIBER EXAMPLE THAT READS DIGITAL DATA ONLY.

Publisher automatically sends a communication cluster that describes the remote system. Use this cluster to properly format your graph and chart setting x and y scales, as shown in the following figure.

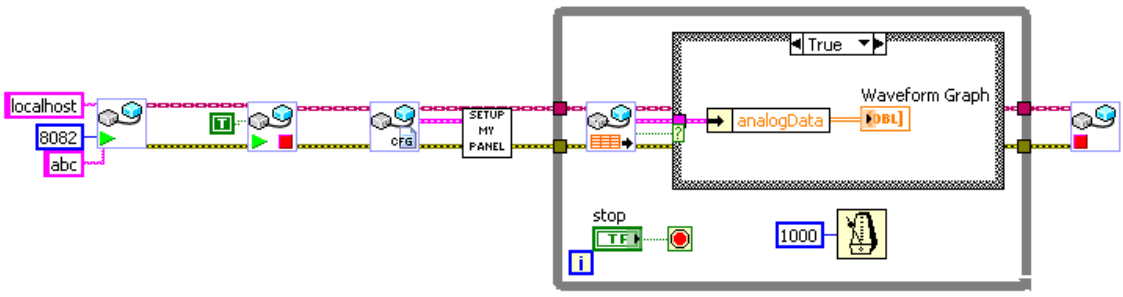


FIGURE 15- SIMPLE SUBSCRIBER THAT READS CONFIGURATION CLUSTER BEFORE READING ANALOG DATA.

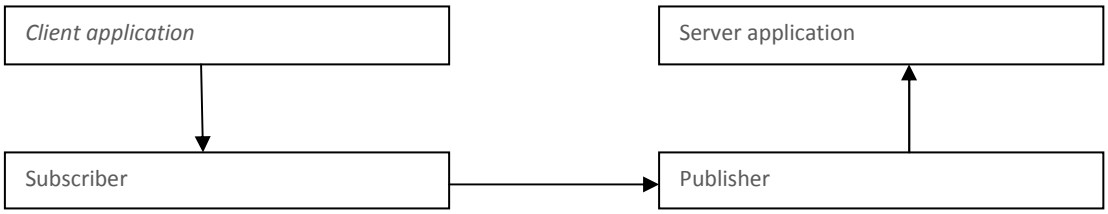
*Notice: **openCommunication.vi** creates a background task which handles the data transfer to and from the publisher. This task is also responsible of checking if connection is lost in case publisher is not sending any information. This task periodically checks if your top level VI is running and stops automatically when your top level VI stops.*

Optionally, you can add an application description to the openConnection.vi. Server side application will use the description to identify your application. This is useful when the server has to properly identify subscriber's name, but they change IP address or port at every new connection, typically when DHCP is used to assign IP addresses. If description is not specified, server will identify the connection with the

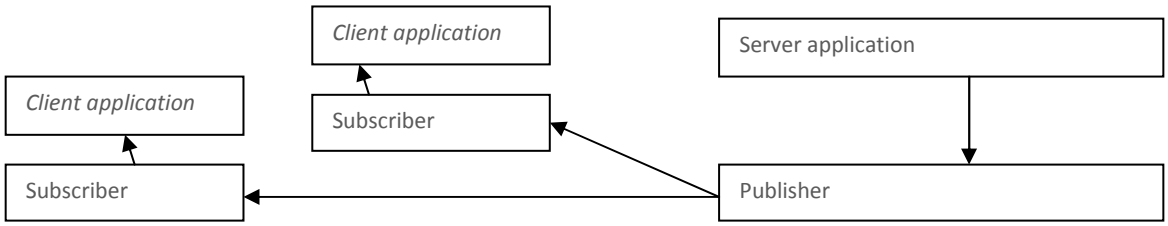
string “xxx.xxx.xxx.xxx:yyyy”, xxx.xxx.xxx.xxx is subscriber application IP address and yyyy is local port used to connect to server.

SENDING MESSAGES

Subscribers can send textual requests to the Publisher with **PublishData_Message.vi.** A message is a cluster composed of a numeric code and a message string. The following figure illustrates how messages are transmitted from a client application to a SCCT server.



Every package regardless of its data type follow the same path indicated in the above figure. When data is transmitted from server to client, it follows the schema indicated below.



The main difference between data transmitted by a client (=subscriber) and a server (=publisher) is that when a client sends a package, this is received by only one application: the server regardless of the number of active connection, instead when a server sends a package, this can be received by all active clients (broadcasting transmission) or by a specific client identified by its IP address.

Messages are sent immediately and receiver processes them in the same order they are received. If message string is empty, message is not posted. You can specify an additional numeric code if your application uses numeric codes to identify message sets.

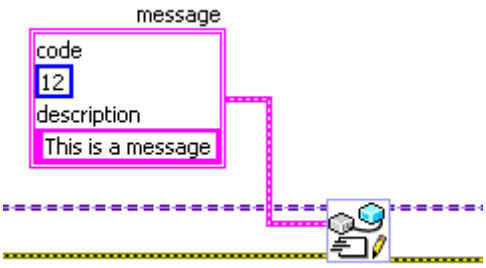


FIGURE 16–PUBLISHDATA_MESSAGE.VI EXAMPLE.

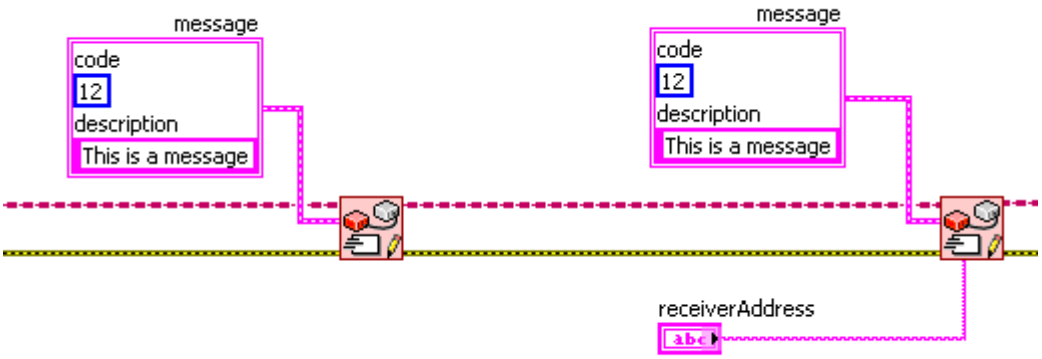
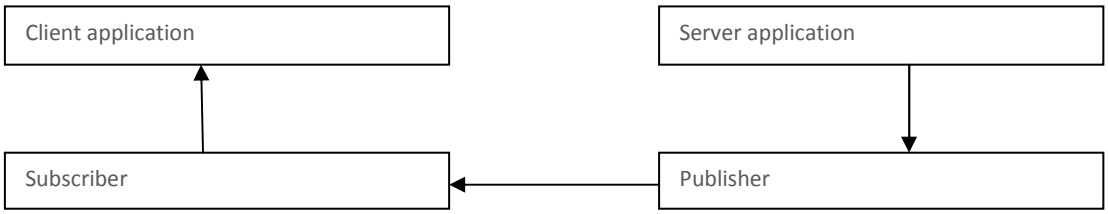


FIGURE 17–MESSAGE SENT BY PUBLISHER CASE IN BROADCAST MODE (LEFT) AND TO A SINGLE CLIENT (RIGHT).

The above figure illustrates two cases where a publisher sends a message: in the first case, message package is transmitted to all active clients, instead in the second case only one client will receive the message. Receiver address is a string composed by client IP address and TCP port. You can retrieve client addresses with **Read activeSubscriberAddresses.vi.**

Receiving Messages

Publisher/Subscribers can receive textual messages with **Read message.vi.** A message is a cluster composed of a timestamp, a string and a numeric code. The following figure illustrates the path of alerts.



Messages are received and enqueued in a dedicated FIFO. Your application has to process incoming messages and take care of FIFO size. You can use Read Message.vi to extract from the message FIFO the oldest received message.

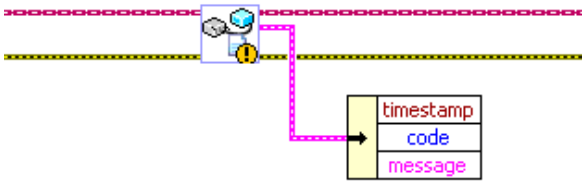


Figure 18- Reading a message from Publisher.

How to check the state of your connection

You can monitor connection state with **connected** property node or **readConnected.vi**, which return TRUE if your connection is still alive. A connection is alive also if Publisher is not sending data to your subscriber. Publisher and subscriber exchange acknowledge packets to verify if connection is still active so you don't have to.

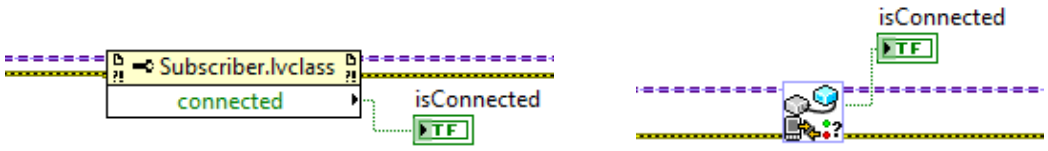


Figure 19- Using properties node and readConnedcted.vi to check connection status.

Closing communication

When connection is no more necessary, use **closeCommunication.vi** to close the open connection. This VI destroys all unprocessed data and closes background tasks. After this Vi, subscriber object cannot be used and a new instance must be created with **openCommunication.vi**.

CREATING AND MANAGING ALERTS AND USER REQUESTS

OVERVIEW

When an active communication is established between publisher and subscriber, they can exchange some messages. Every message is composed of a numeric code and a message string. Every message contains a numeric code, a timestamp, a connection identifier, an event code and an optional data string. Publisher uses a FIFO to enqueue all incoming messages in the order they are received. Your application can identify which subscriber is sending the message through its ipAddress:port identifier.

READING AVAILABLE MESSAGES

Subscribers send their request to your application and Publisher keeps them in a FIFO together with some messages it sends to inform your application about the connection status and the communication between publisher and subscribers. Use **availableRequests** property node to retrieve the number of received request that your application has to process. When a request is processed with **getRequest.vi**, **availableRequests** is decremented by 1.

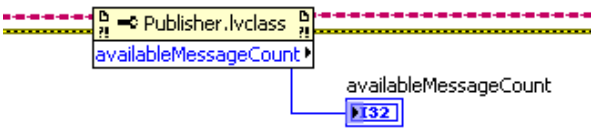


FIGURE 20- USING *AVAILABLE MESSAGE COUNT* PROPERTY NODE TO CHECK IF SUBSCRIBER'S MESSAGES ARE PRESENT.

SENDING MESSAGE TO A SPECIFIC SUBSCRIBER

Your application can communicate with active subscribers with custom messages. To send a message use **PublishData_Message.vi**, as shown in the following figure:

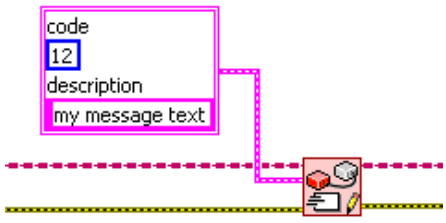


FIGURE 21- SENDING MESSAGE TO A SPECIFIC SUBSCRIBER WITH PUBLISHDATA_MESSAGE.VI.

Use **message** string to add additional information to your message. To send a message to every active subscriber **receiverAddress** must be empty string, instead, if you want to send a message to a specific subscriber, connect **receiverAddress** to the specific address and port of its connection. Message string cannot be empty. If try to send an empty string message, error is generated with error code 5002.

MANAGING USER MESSAGES

User message (i.e. message from subscriber) is inserted into a FIFO by Publisher so your application can process allmessages in the order they are received from Publisher. The following figure illustrates the right way to manage the messages:

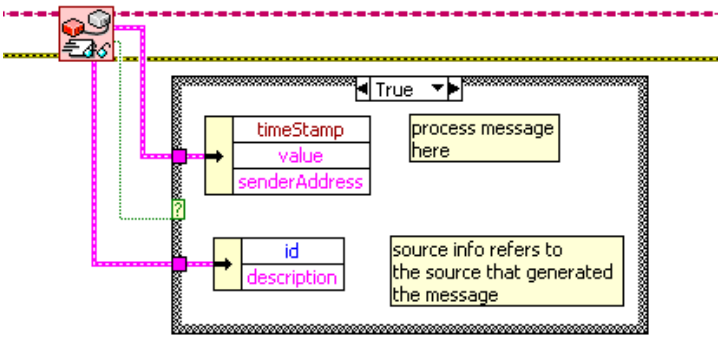


FIGURE 22- MANAGING RECEIVED MESSAGE EXAMPLE.

First, you have to check that **found** indicator is TRUE, if **found** is FALSE, no message is available. Every message has its **time** field, **senderAddress** field that identifies the subscriber that sent the message. If connected client manages multiple data sources, **source** indicator specifies which source has generated the message

NOTIFYING AN ERROR

When a LabVIEW error arises and must be notified to one or more subscribers, use notifyError.vi which composes a message with error code and error description from LabVIEW error cluster. The following example shows how to use it. Notice that you can specify a ipAddress:port reference to send error only to a specific listener. When error cluster contains no error, no message is sent.

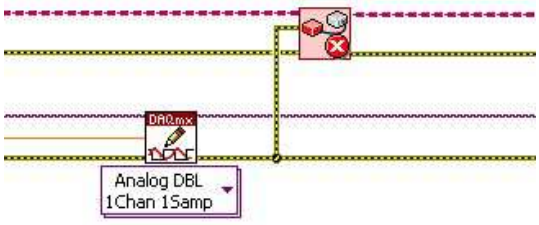


FIGURE 23- NOTIFYING LABVIEW ERROR TO ALL SUBSCRIBERS WITH NOTIFYERROR.VI.

SOURCE SELECTION

To reduce data traffic, SCCT includes a feature called *source selection* that can significantly reduce the used band on your communication channel. It applies in cases where server manages multiple data sources (every data source is identified by its ID number, a 32bit integer) and clients need to receive data from a subset of these sources, at the same time. The following figure illustrates the case you have a server with 4 acquisition devices and you client app needs data from 2 devices at the same time.

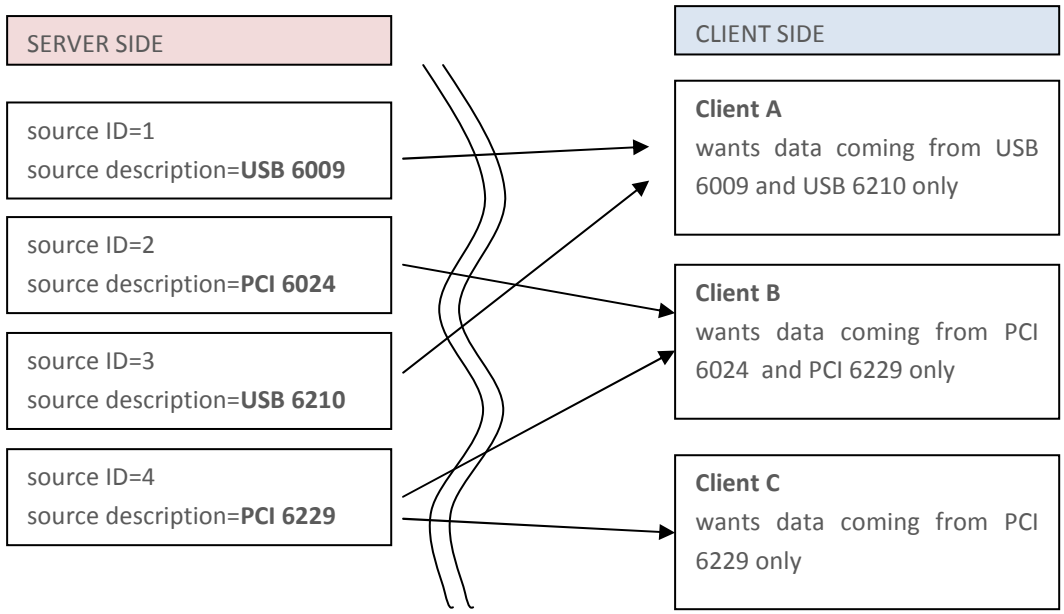


FIGURE 24 – SCENARIO WITH A MULTIPLE DATA SOURCE SERVER AND CLIENTS THAT NEED A SUBSET OF PUBLISHED DATA.

In scenarios like the one above, you have to consider source selection at server side. Use the following schema to improve your communication performance and reduce used band between server and client.

STEP 1 - CONNECTION WITH FILTER ENABLED
Client connects to server with filterSelectionEnabled=True option After connection is established, server has an empty list of selected sources so no data packages are sent to client.
STEP 2 - SERVER INFORMS CLIENT ABOUT AVAILABLE SOURCES
Your server has to publish a package that contains a list of available data sources (usually it is part welcome kit packages).
STEP 3 - CLIENT SENDS ITS DATA SOURCE SELECTION
At client side, your application shows source list and user chooses which sources wants to use. Client

sends a selected source list package, with its specific VI.
STEP 4 - SERVER SETS UP SOURCE FILTER (AND WAITS DATA TRANSFER=TRUE)
Server receives the selected source list package and updates the client's list so data packages are filtered properly.
STEP 5 - CLIENT ENABLES DATA TRANSFER
Client app sends start=true with dataTransfer.vi and enables data transmission.

Selected source list can be changed at any time during connection so clients can switch among available data sources without closing existing connection.

ENABLING SOURCE FILTER

Source filter is enabled at connection time so your subscriber has to inform the server that is going to take advantage of source selection filter to reduce data traffic. Specify **sourceFilterEnable=True** when try to open a connection as show in the following figure:

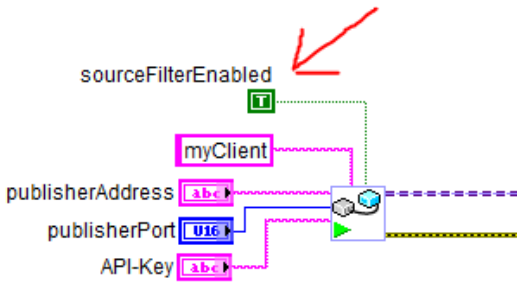


FIGURE 25- SET SOURCEFILTERENABLED=TRUE WHEN YOU PLAN TO TAKE ADVANTAGE OF DATA FILTERING AT SERVER SIDE.

By default sourceFilterEnabled is set to false, so server sends all data packages regardless of their source.

SELECTING SOURCE LIST

Use **selectSourceList.vi** to select the sources your client wants to receive. This command has only if you specified **filterEnabled=true** at openConnection.vi. Before you select one or more sources, server keeps its source list empty so no data packages are sent. The following figure illustrates the right way to select a source list:

FIGURE 26- SOURCE LIST SELECTION.

Source list can be selected before the connection start or during the connection session. Your client has to know the source ID and description list. Your server has to inform clients about available sources. Source list can be published with different type packages: message, custom XML Data, customData. Define a source list so that clients can decode it easily.

SENDING A FILTER COMMAND

As mentioned before, subscribers can use **selectSourceList.vi** in order to receive data from specific sources only and reduce the amount of transmitted data. Moreover, SCCT allows to select a portion of data. For example, if a subscriber needs only a subset of available published signals, it can request the channels it needs through **FilterRequest.vi**. In this way, for each selected source, subscribers can request to receive only a portion of data. Notice that with both filtering and source list, SCCT performs a server-side filtering.

Filtering is available for analog data, 2D array of booleans, strings, I16, I32, I64, DBL, and SGL data. For analog data, filtering allows to select specific channels. Otherwise, for 2D arrays, filtering permits to select a subset of the available published array data. The following figure shows how to send a filter command to server for different data types.

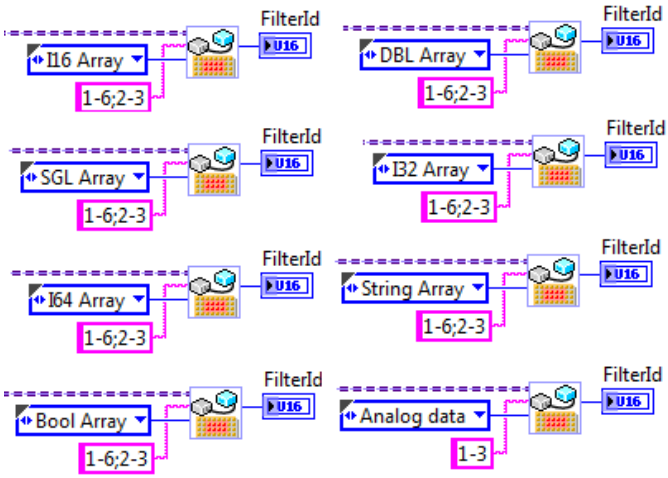


FIGURE 27 – SENDING A FILTER REQUEST TO SERVER.

A different FilterId is generated for every filter request. When server performs a new filtering operation, it increments the FilterID field in same way it is modified by the client. In this way, server and client maintain a counter in common. The FilterID permits the client to relate every received data package to the relative filter request. FilterId is fundamental to process received data in the right way. For example, if server manages a 100x100 array and a client needs a 11x11 subarray (from index 5,5 to 15,15), the client sends a filter request. As it is the first filter request sent to the server, the filter request must generate a FilterId = 1. When the filter request is performed, server sends the required subarray and marks transmitted packages with FilterID = 1. Later, if the client needs a different 11x11 subarray, it sends a another filter request. As it is the second filter request, the filter request generates a FilterId = 2. The server performs this new filter request, sends the new subarray and marks transmitted packages with FilterId = 2. The following figure schematizes this example. The use of FilterId permits the client to link every received package with the first or the second filter request.

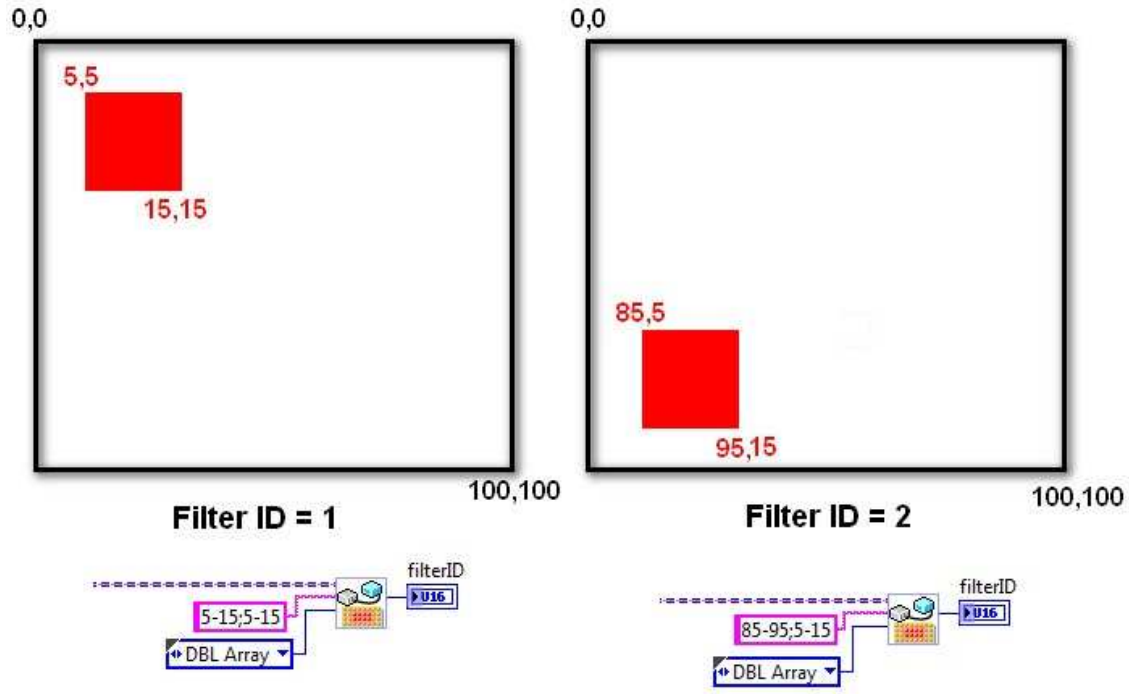


FIGURE 28 – SENDING TWO DIFFERENT FILTER REQUESTS TO SERVER.

The FilterId of every data package is stored in the **additionalInfo** cluster. Use the following code to read the FilterID

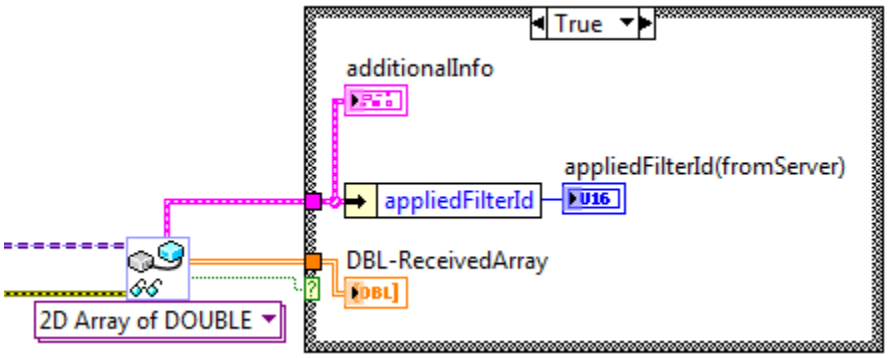


FIGURE 29 - READING FILTERID FOR EVERY RECEIVED DATA PACKAGE.

MANAGING BROADCASTING MESSAGES TO BY-PASS SELECTED SOURCE FILTER

In previous paragraphs we have learned that we can filter data packages at server side and server doesn't send data packages if they don't match the source list filter. source filter can be bypassed by broadcasting transmission. when server has to notify special packages (for example alarm messages or server state changes) you can set Broadcast flag to True and your package regardless of data type, is sent to all active connections.

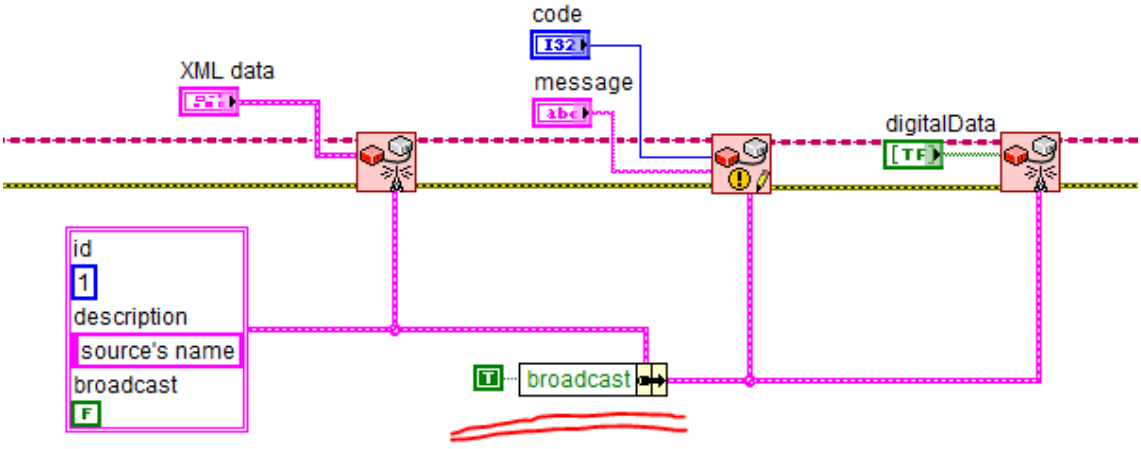


FIGURE 30- BROADCAST TRANSMISSION.

The previous example illustrates how to force broadcast transmission. First package is sent to clients who added source with ID=1 to their source list. Second and third package are sent to all clients regardless of their source filter. Notice that when broadcast flag is set to True, source ID and description are ignored by transmission task.

PUBLISHING AND RECEIVING IMAGES

SCCT 2.1 allows clients and server to exchange images. Images are passed with a list of attributes assigned at run time by user applications. The following examples illustrates how to send and receive image with a publisher, the same concepts can be used to send and receive images with a client.

PUBLISHING AN IMAGE (EXAMPLE 1)

In this example is shown the case a server publishes images to all connected clients.

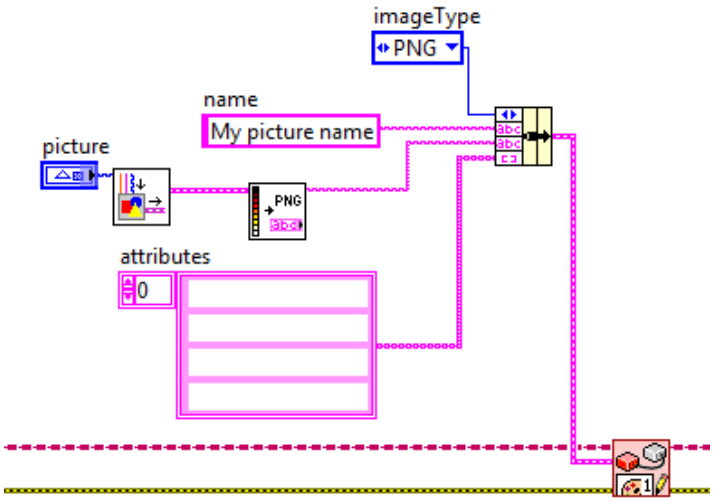


FIGURE 31– PUBLISHING AN IMAGE WITH PUBLISHDATA_IMAGE1.VI.

The above example regards the case your image is in memory and is represented by **Image data** cluster. **PublishData_Image1.vi** takes as input values, a cluster containing image type, image name, image plan data, in its native format and a list of image attributes. Image has to be passed in its native format and not as matrix of pixel. Use **PublishData_Image1.vi** when your images are in memory or produced by some image processing function and you don't need to save them on disk.

PUBLISHING AN IMAGE (EXAMPLE 2)

The following example illustrates the case a server publishes images saved on disk. SCCT can load and publish images with a single function, as shown below. When images you need to publish are stored on disk and you don't need to display them in your application, use **publishData_Image2.vi** to transmit selected image to all connected clients. Every image can be described by a name and with a list of attributes.

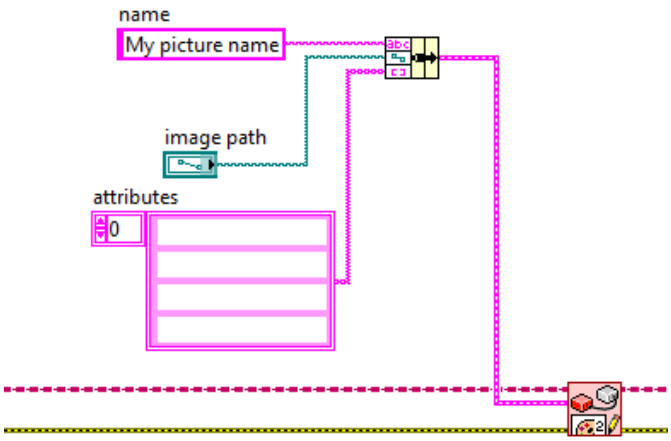


FIGURE 32– PUBLISHING AN IMAGE STORED ON DISK, WITH PUBLISHDATA_IMAGE2.VI.

RECEIVING AN IMAGE (EXAMPLE 1)

In this example is shown the case server receives an image from a client. The strategy used to receive data from connected clients is always the same for all data type. Read Image.vi returns a Boolean indicator to indicate if images are available. Is at least one image is available, it is

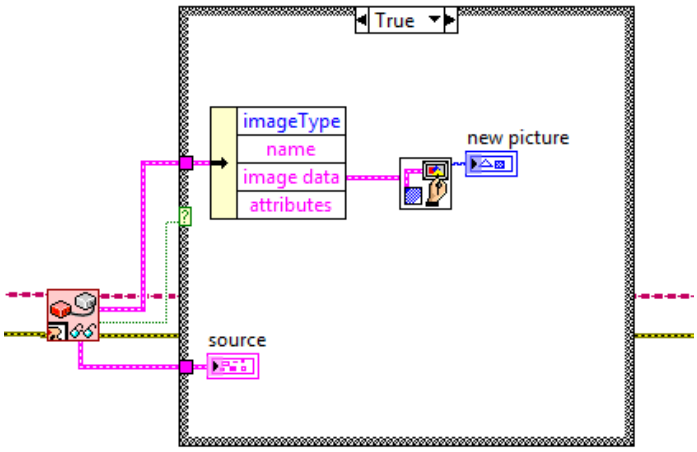


FIGURE 33– RECEIVING AN IMAGE FROM A CLIENT WITH READ IMAGE.VI.

PUBLISHING AND RECEIVING 2D ARRAYS

SCCT permits clients and server to publish and receive 2D arrays of different data types. The supported data type are array 2D of booleans, strings, word signed integers (I16), long signed integers (I32), quad signed integers (I64), double-precision floating-point (DBL), and single-precision floating-point (SGL).

The following examples illustrates how to publish and read 2D arrays. Array data are published through *publishData_BOOL_Array.vi*, *publishData_I16_Array.vi*, *and publishData_string_Array.vi*; otherwise, you can use the polymorphic VI *publishData.vi*. You can read arrays through *read.vi* that is a polymorphic VI.

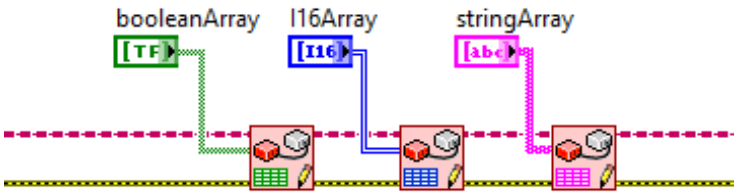


FIGURE 34 - PUBLISHING DIFFERENT TYPES OF 2D ARRAYS

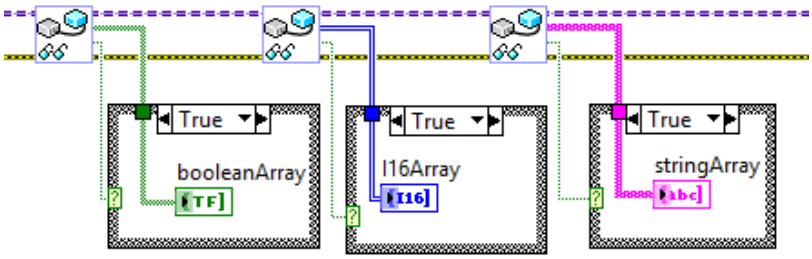


FIGURE 35 - READING DIFFERENT TYPES OF 2D ARRAYS

PUBLISHING AND RECEIVING FILES

SCCT can enables clients and server to exchange images. Images are passed in their native formats with a list of attributes assigned at run time by applications.

FILE EXCHANGE EXAMPLE 1

In this example is shown the case a server publishes files to all connected clients. files are loaded from a folder and transferred to clients with some attributes specified at application level.

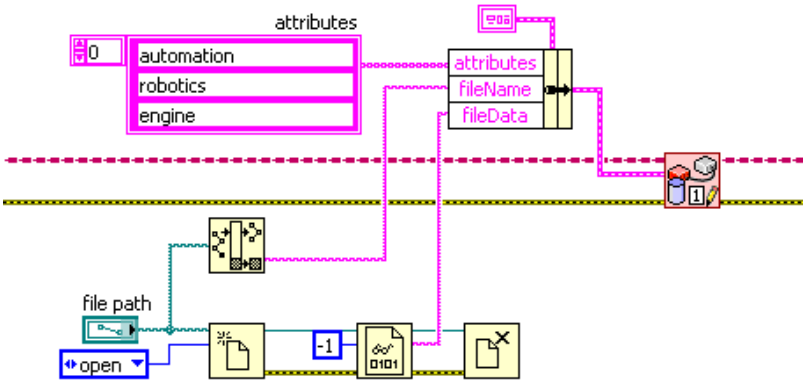


FIGURE 36– PUBLISHING A FILE WITH PUBLISHDATA_FILE_1.VI

In the above example file is loaded at application level and sent with three attributes.

FILE EXCHANGE EXAMPLE 2

Files can be sent in a very way when their full path is know. The following example sends a file and set **computeMD5**flag to Trueso that SCCT computes MD5 value and associate it to the transmitted file.

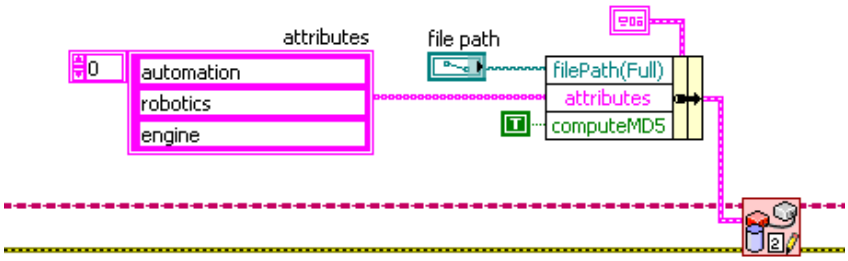


FIGURE 37– PUBLISHING A FILE WITH PUBLISHDATA_FILE_2.VI

Remember that SCCT validates every transmitted and received package regardless of package data type so MD5 value is not necessary to clients to validate integrity of received files.

ADVANCED OPTIONS

This chapter illustrates advances features of SCCT for LabVIEW.

CONTROLLING ACTIVE CONNECTIONS

Publisher Vis handle all incoming connections and close inactive connections so you just focus on your main application and forget all issues related to data transmission. In some cases you want to know the number of active connections and the address of subscribers. Usually you can map all incoming

connections using getRequest.vi and filtering events such “connection successful” and “connection closed” and “connection timeout”, which help you to map all active and closed connections.

READING ACTIVE CONNECTION COUNT

To know the count of active connections at a specific time, use **Read activeSubscribersCount.vi** that returns the number of active connections.

FIGURE 38- USE READ ACTIVESUBSCRIBERSCOUNT.VI TO CHECK THE COUNT OF ACTIVE CONNECTIONS.

READING ACTIVE CONNECTION ADDRESSES

To know the addresses of active subscribers, use **Read activeSubscribersAddresses.vi that** returns a string array: every string is a subscriber’s address in the form ipAddress:port. the array is ordered by connection time so the first element of the array is related to the active subscribers that connected first. Closed connections don’t appear in the array.

FIGURE 39- USE READ ACTIVESUBSCRIBERSADDRESSES TO GET INFO ABOUT ACTIVE SUBSCRIBERS.

You can use **activeSubscriberCount** property node to get active connection count and **activeSubscriberAddress** property node to get active connection addresses, as shown below:

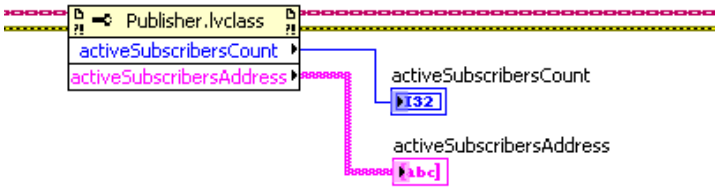


FIGURE 40 - USE PROPERTY NODES TO GET INFORMATION ABOUT ACTIVE SUBSCRIBERS.

READING ACTIVE SUBSCRIBERS PROFILES

At server side you can get detailed information about active connections. Every connection is identified by client IP address and port number. Use address:port as search key to retrieve subscriber's profile. The client profile is composed of:

- Start connection time. It indicates time and date when client has established connection.
- Description. It contains the description of the client application, specified when client has connected to server.
- Platform. It indicates the client platform: LabVIEW, Android, iOS, HTML5.
- Release. It indicates the SCCT release.
- Connection ID. It is an univocal code generated for every active connection.
- Selected Sources. It is a list of the selected sources by the client.
- Data transfer status. It indicates the current data transfer status of the client (enabled or disabled).
- Source Filter status. It indicates the current source filter status of the client (enabled or disabled).
- API- Key. It is the API-Key used by the client to connected with server.
- Timeout. It represents the timeout of the client.

The following example illustrates the case you search user profile who establishes a new connection.

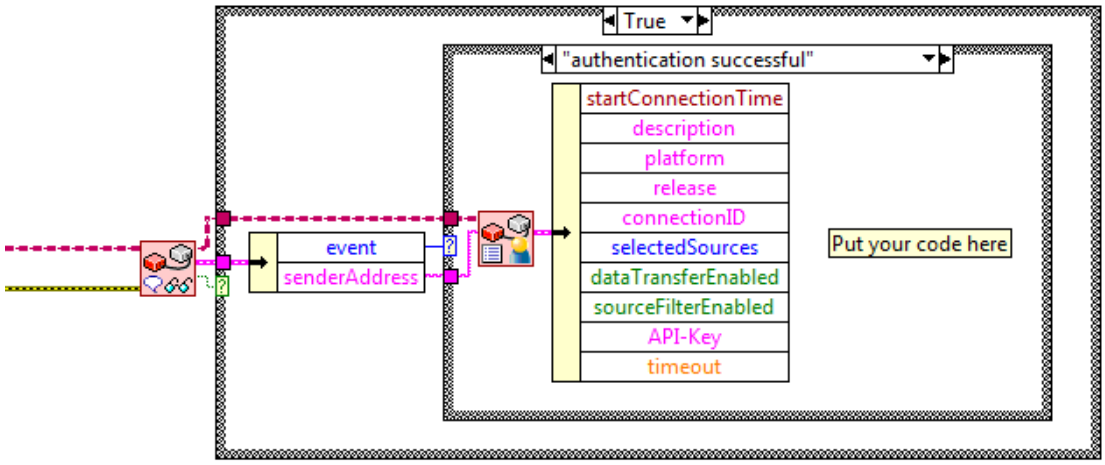


FIGURE 41- USE READ SUBSCRIBERPROFILE.VI TO RETRIEVE INFORMATION ABOUT AN ACTIVE CONNECTION.

When a connection terminates, its data are removed immediately form publisher internal database so you cannot retrieve its parameters. If Read subcriberProfile.vi returns an empty cluster, means that the connection has been closed after the request has been queued.

MANAGING EVENTS

Publisher provides a FIFO used to keep track of communication events. This FIFO is enabled by default and can be enabled / disabled with **eventLogging** property node (see above). Event data cluster contains the following information:

- **timestamp** is the local time when event is occurred.
- **Event** describes the type of event. The available event types are:

- **authentication successful**, occurs when a new connection is properly established. Is a remote device tries to connect, but its connection parameters are not correct and connection is refused, this event is not generated.
- **conn closed by client,** occurs when a client terminates connection.
- **timeout expired,** occurs when a connection is lost abnormally, for example when communication channel fails. In this situation your application cannot be sure that all transmitted packets have been received by client associated to aborted connection.

- **senderAddress** is the address of new subscriber, in the form IpAddress:port.

Use **availableEventCount** property node to check is new events have been generated. When a new event is present you can extract it from its event Queue with **getEvent.vi** method as shown in the following figures.

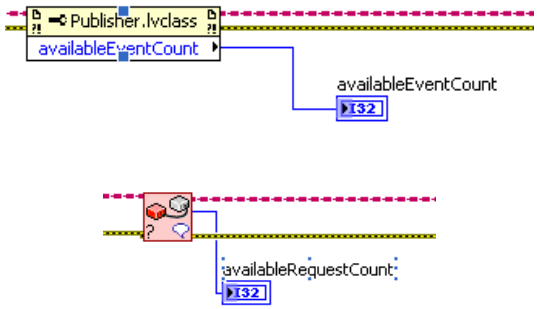


FIGURE 42 – ENABLING THE REQUEST FIFO WITH ENABLEEVENTLOGGING PROPERTY NODE..

As shown above, instead of property node you can set the flag “enableLogging” with **enableLogging.vi** and get its value with **isEventLoggingEnabled.vi**, respectively

Events are useful when your application keeps track of existing connections and must execute special tasks when a new connection is established or an existing one is closed. Below you see a simple example: a welcome message is sent only to new connected applications. **senderAddress** is the address associated to the connection that generated the event.

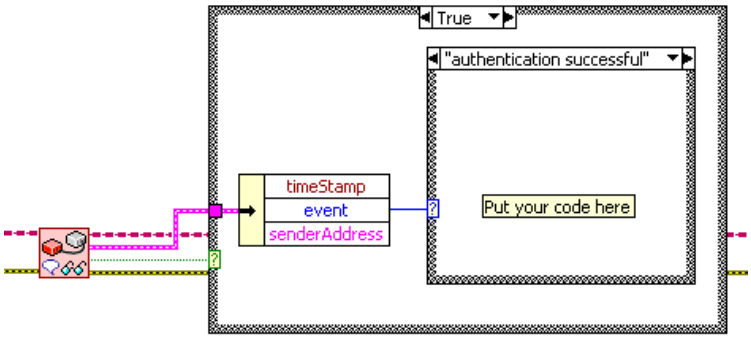


FIGURE 43 – THIS EXAMPLE SHOWS HOW TO MANAGE "AUTHENTICATION SUCCESSFUL" EVENT.

In the following figure a different situation is managed: when a connection is terminated, and your application has to do some action like deleting temporary files associated to the connection.

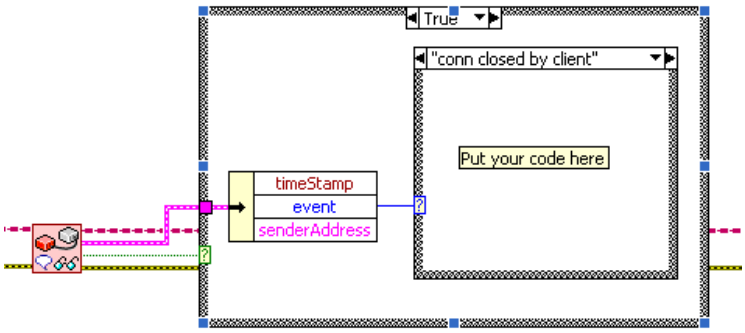


FIGURE 44 – THIS EXAMPLE SHOWS HOW TO MANAGE THE CASE A CONNECTION HAS BEEN CLOSED BY CLIENT.

CHANGING API-KEY AT RUN-TIME

Publishers are created with an API-Key they use to authenticate every incoming connection Notice that SCCT doesn't accept null API-KEY values. When publisher is started you can add and remove API-KEYs programmatically using **Add API-KEY.vi** and **Remove API-KEY.vi**. Publisher can use unlimited number of API-KEYs and when a subscriber tries to connect and provides its API-KEY, publisher search that KEY among the API-KEY set created at run time. You can also remove existing API-KEY except when you have only one KEY: publisher aborts KEY removal when it has only one KEY.

ADDING A NEW API-KEY

The following example illustrates the case you want to add an API-KEY to the active set of KEYs. Existing connections are not affected by this command.

FIGURE 45 – ADDING API-KEY TO ACTIVE API-KEY SET.VI.

Adding API-KEYs can be done with **Add API-KEY.vi** or a property node as shown above.

REMOVING AN EXISTING API-KEY

The following example illustrates how to remove existing API-KEY from active set of KEYs. Existing connections are not affected by API-Key changes.

FIGURE 46 – REMOVING AN API-KEY FROM ACTIVE API-KEY SET.VI.

Adding API-KEYs can be done with **Remove API-KEY.vi** or a property node as shown above.

CHANGING MAX DT

When subscribers try to connect, the request includes their system time. If subscriber's system time differs from publisher's system time more than **maxDt** and **maxDt** is greater than zero, connection is refused. By default, **maxDt** value is zero so subscriber's system time is not checked. To enable time control, use **maxDt** property node to set the absolute value of maximum distance, in seconds, between local and remote system's time.

FIGURE 47 – CHANGING MAXDT TO CHECK SUBSCRIBER'S SYSTEM TIME.

WRITING AND READING SAFETY MODE

Packages sent with SCCT are enqueued in a FIFO structure. If data are produced at higher rate than data processing speed, or if connection bandwidth is limited with respect to data production rate, the number of pending packages and memory requirement of server application increase. To avoid this problem, SCCT permits to enable Safety Mode. When safe mode is enabled, no more than 100 packages

can be pending (awaiting to be transmitted) for every connection. Exceeding packages are discarded. To enable/disable safety mode, use Write safetyModeEnabled. By default, Safety Mode is disabled. To read Safety Mode, use read safetyModeEnabled.

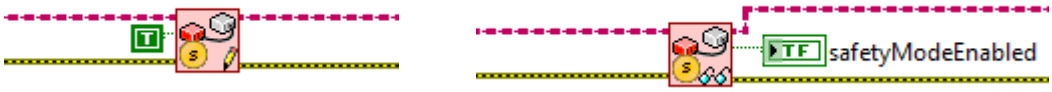


FIGURE 48 – WRITING AND READING SAFETY MODE.

PUBLISHING AND RECEIVING CUSTOM DATA (SUPPORTED ONLY BY LABVIEW APPLICATIONS)

If both side applications use SCCT for LabVIEW, then some advanced features are available. These features rely on LabVIEW variant data type and their data package are not supported on other programming languages. if more than one custom data type is used, custom data types must be associated to a numeric code or string description so that subscribers can properly identify what data has been sent from publisher. In the following example, publisher side, on the left, encode two different data types and send them with**publishData.vi**. Notice they are associated to code 1 and 2. Subscriber application, on the right, receive the packages with **Read customData.vi**, and uses code field to properly process data type 1 and data type 2.

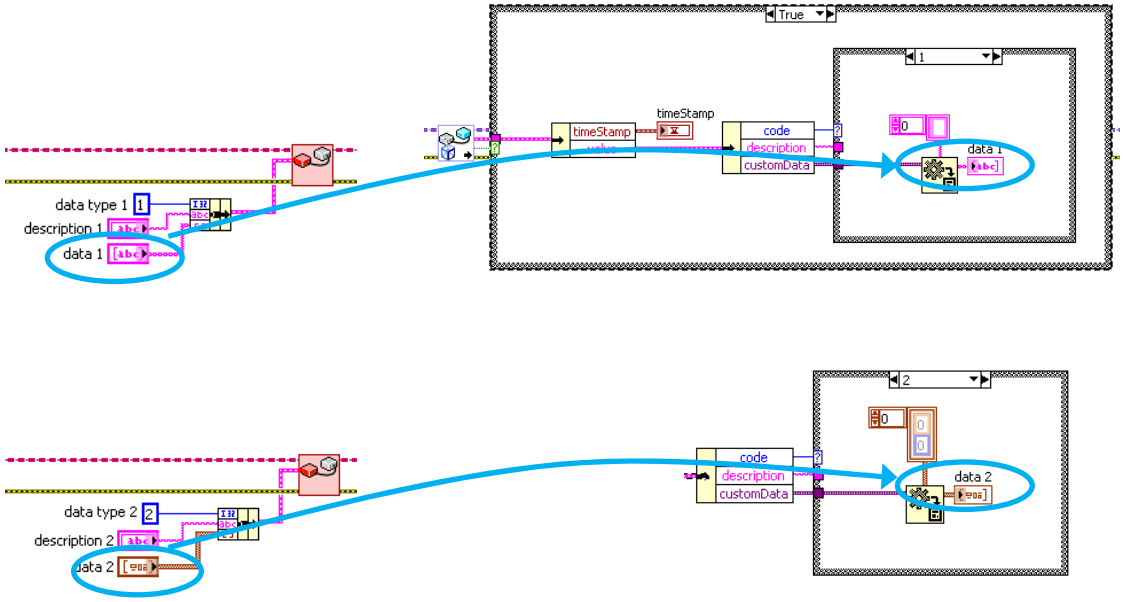


FIGURE 49–PUBLISHER APPLICATION AND SUBSCRIBER APPLICATION USE CODE FIELD TO IDENTIFY THE PROPER DATA TYPE. BOTH APPS HAVE TO KNOW THE DATA TYPE ASSOCIATED TO THE CODES.

PUBLISHING AND RECEIVING CUSTOM XML DATA

In addition to messages and requests, both sides of communication channel can exchange XML documents. XML packets have been added to SCCT to allow server and client applications exchanging complex custom data with a single function. Remember that XML documents are not check by SCCT Vis. You must verify XML document's correctness before transmission to avoid unpredictable behaviors by receiving applications. The following figures illustrate the available set of Vis in Publisher and Subscriber's libraries. Notice that on Publisher side.XML transmission is done with the Polymorphic VI **PublishData.vi.**

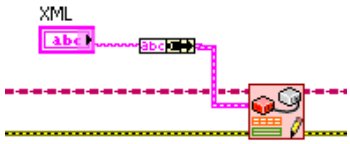


FIGURE 50 – WITH PUBLISHER, USE PUBLISHDATA.VI METHOD TO SEND XML DOCUMENTS.

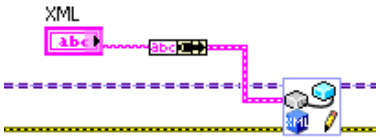


FIGURE 51 – WITH SUBSCRIBER, USE SENDCUSTOMXMLDATA.VI TO SEND XML DOCUMENTS TO PUBLISHER.

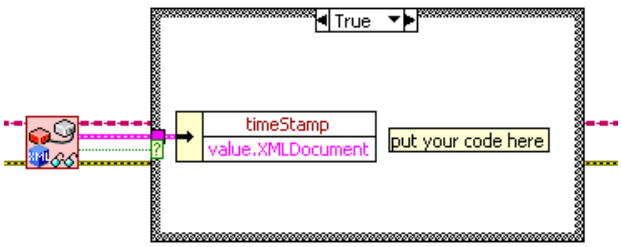


FIGURE 52 – WITH PUBLISHER, USE READ CUSTOMXMLDATA.VI TO PROCESS XML PACKETS.

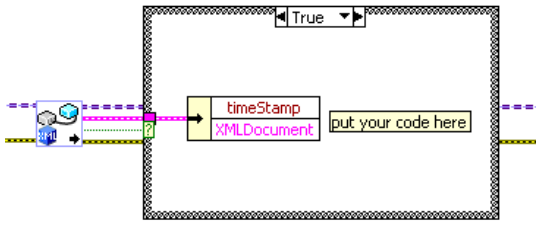


FIGURE 53 – WITH SUBSCRIBER, USE READ CUSTOMXMLDATA.VI TO PROCESS XML PACKETS.

Publisher and Subscriber manage XML document transfer in the same way.

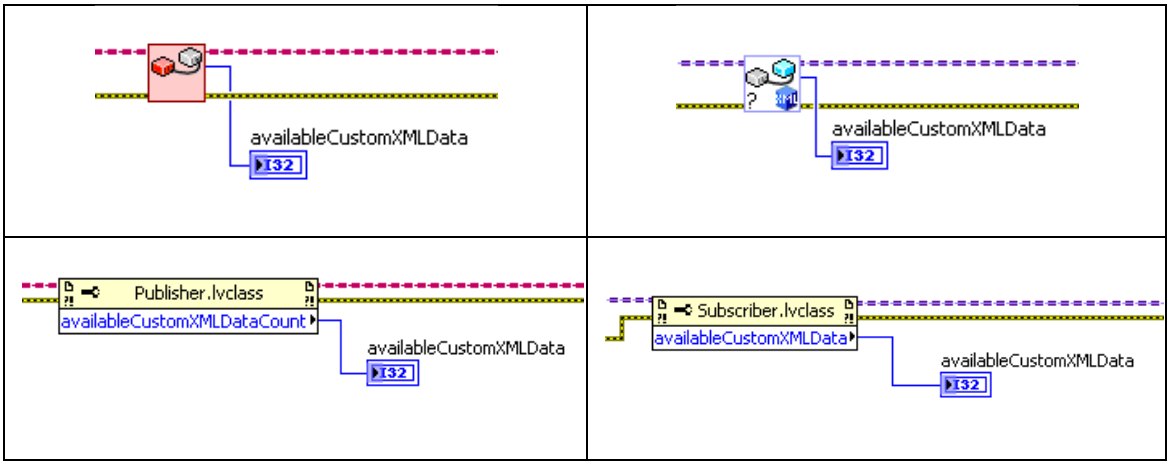


FIGURE 54 – WITH PUBLISHER AND SUBSCRIBER YOU HAVE DIFFERENT METHODS AND PROPERTY NODES TO GET THE COUNT OF AVAILABLE XML UNPROCESSED DOCUMENTS.

PUBLISHING AND RECEIVING SERVER LOCATION

When publisher) is running on a mobile system (for example on a vehicle) and it is provided of a GPS or similar localization system, it can update its current location to subscribers with **updateLocation.vi**. At client side, subscribers can manage location packages in the same way they handle other package types, with **Read availableLocationCount.vi** and **Read newLocation.vi**. The following examples illustrate how to integrate location management in your applications.

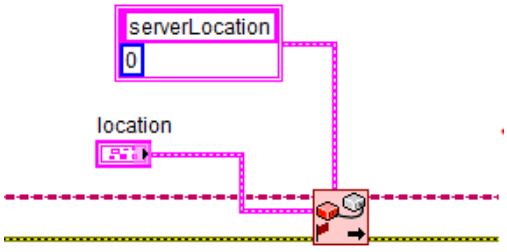
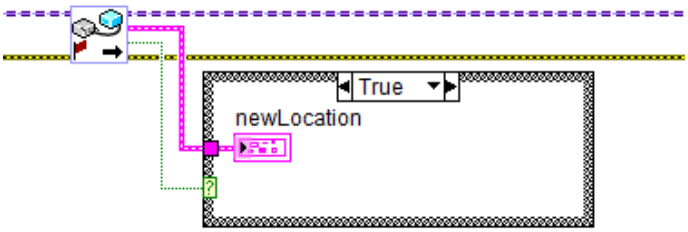


FIGURE 55 – USE UPDATELOCATION.VI TO SEND CURRENT PUBLISHER POSITION TO CLIENTS.



Use the same programming approach used to handle incoming data packages, check found indicator before processing location packages. Use **Read availableLocationCount.vi** to retrieve to number of unprocessed location packages.

FIGURE 56 – IN SUBSCRIBER, USE SENDCUSTOMXMLDATA.VI TO SEND XML DOCUMENTS TO PUBLISHER.

FIGURE 57 – IN SUBSCRIBER, USE READ AVAILABLELOCATIONCOUNT.VI TO VERIFY THE NUMBER OF UNPROCESSED LOCATION PACKAGES.

BUILD SPECIFICATIONS

Application with Subscriber VIs

Creating executables that include VIs from Smartphone & Cross-platform Communication Toolkit requires a specific setting of "Additional Exclusions" tab page. flags from "Modify project library file after removing unused members" must be unchecked, as shown in the following figure.

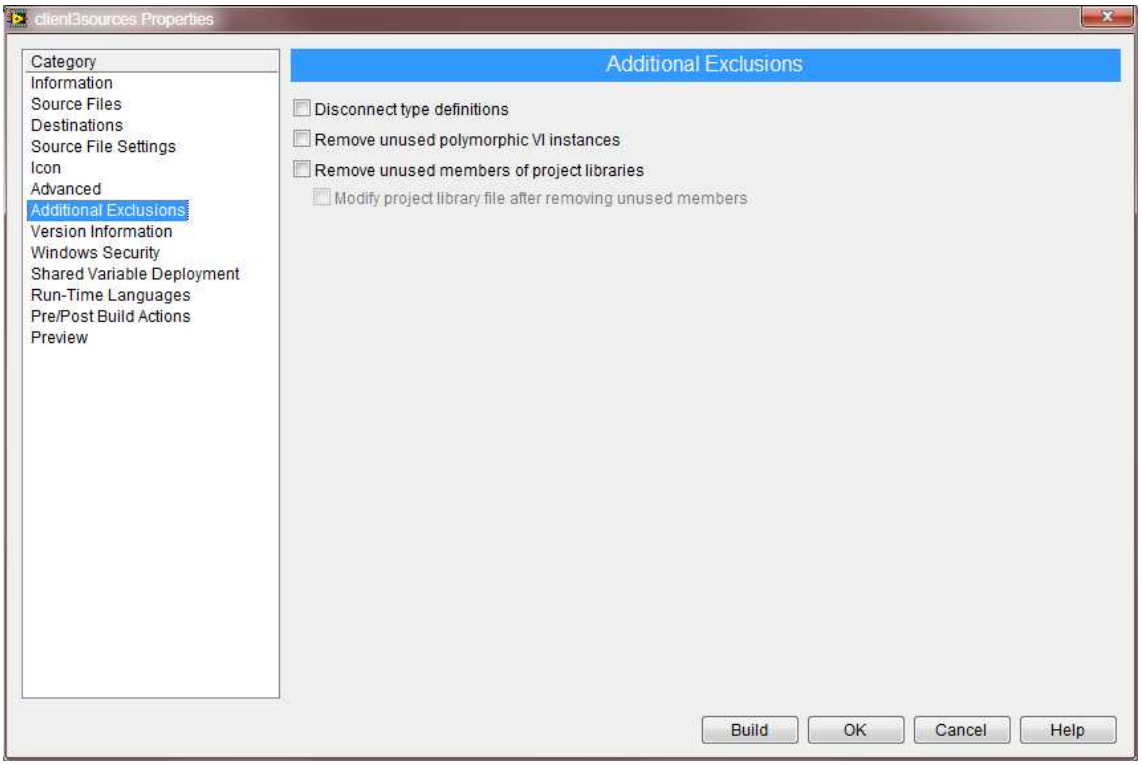


FIGURE 58 – "ADDITIONAL EXCLUSIONS" PAGE MUST BE PROPERLY CONFIGURED.

Smartphone & Cross-platform Communication Toolkit is password protected so if above additional exclusion is not unchecked, the following error stops building process:

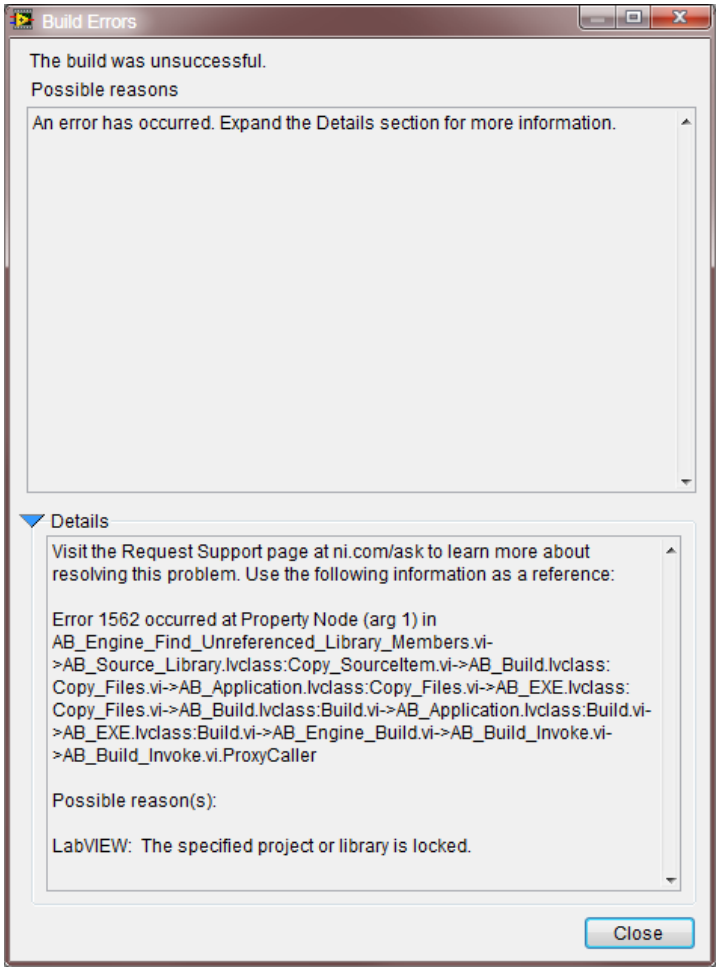

Build Errors
The build was unsuccessful.
Possible reasons
An error has occurred. Expand the Details section for more information.
Details
Visit the Request Support page at ni.com/ask to learn more about resolving this problem. Use the following information as a reference:
Error 1562 occurred at Property Node (arg 1) in AB_Engine_Find_Unreferenced_Library_Members.vi->AB_Source_Library.lvclass:Copy_SourceItem.vi->AB_Build.lvclass:Copy_Files.vi->AB_Application.lvclass:Copy_Files.vi->AB_EXE.lvclass:Copy_Files.vi->AB_Build.lvclass:Build.vi->AB_Application.lvclass:Build.vi->AB_EXE.lvclass:Build.vi->AB_Engine_Build.vi->AB_Build_Invoke.vi->AB_Build_Invoke.vi.ProxyCaller
Possible reason(s):
LabVIEW: The specified project or library is locked.
Close

FIGURE 59 – "BUILD ERRORS" WITH ERROR CODE 1562.

APPENDIX A – ERROR TABLE

In the following tables are indicated the error codes generated by VIs

PUBLISHER ERROR CODES

code	description	explanation
5000	Publisher is not started	A method or property has been called before creating the Publisher
5001	Invalid port	When you start the publisher you have to provide a valid TCP port.
5002	invalid message text	You cannot send a message with an empty text message
5003	Invalid API-Key	API-Key cannot be an empty string
5004	Invalid configuration	Configuration cluster must contain at least 1 analog channel or 1 digital line
5005	Invalid analog data	Analog data must be a 2D array with exactly the number of rows equal to the number of analog channels as defined in configuration cluster.
5006	Invalid digital Data	Digital data must be a 1D array with size equal to the number of digital lines as defined in configuration cluster
5007	Empty data are not published	Empty arrays are not published
5008	internal library is corrupted	Unexpected error during creation of communication tasks

SUBSCRIBER ERROR CODES

code	description	explanation
6000	Connection is not established	A method or property has been called before establishing the connection or it has been closed
6001	Invalid port	When you start the publisher you have to provide a valid TCP port.
6002	invalid timeout	Timeout must be a value greater than zero
6003	Invalid API-Key	API-Key cannot be an empty string
6004	Connection cannot be created twice	openConnection.vi cannot be used on an active connection. You have to close a connection before re-open
6005	Connection refused from Publisher: wrong API-Key	Publisher refused the connection because API-Key is not correct
6006	Publisher not found	No answer from specified address: possible reasons: publisher is down or not reachable
6007	Publisher Address cannot be empty	Publisher Address cannot be an empty string

INDEX

A

active connection

addresses 47

count 47

Analog data

Publish 25; 26

Read 32

API-Key

changing 50

API-KEY

adding 50

removing 51

Arrays 45

authentication successful 49

available Requests 36

C

Communication

Close 35

Open 32

Custom data

Publish 53

custom XML data 54

D

data transmission

start 31

stop 31

DHCP 32

Digital data

Publish 27

E

Error

Notifying an error 37

error codes

Publisher 59

Subscriber 60

Event

authentication successful 48

connection closed by client 48

managing 48

timeout expired 48

F

Files publishing and receiving 45

Filter

Server-side filtering 40

Filtering

FilterId 41

I

Images publishing and receiving 43

M

Messages

Receiving 35

P

Publisher

Creating 24

methods 23

properties 23

Publisher Class 23

Publishing

Analog Data 25

Custom Data 53; 54; 55

Digital Data 27

Request 33

R

receiving

XML 54

Requests

Sending 33

S

sending

XML 54

Source List

broadcast messages 42

Source selection 38

filter enabling 39

selecting 39

start

data transmission 31

stop

data transmission 31

Subscriber

Creating 31

methods 29

properies 29

Read Profile 47

Subscriber Class 29

T

timeout expired 49

U

User Request 37

W

Welcome Kit 26

X

XML 54

receiving 54

sending 54